ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

Herramienta para la generación automática de casos de prueba mediante siembra automática para WS-BPEL 2.0

Curso 2013-2014

Valentín Liñeiro Barea

Cádiz, 4 de febrero de 2014

ESCUELA SUPERIOR DE INGENIERÍA

Segundo Ciclo en Ingeniería Informática

Herramienta para la generación automática de casos de prueba mediante siembra automática para WS-BPEL 2.0

DEPARTAMENTO: Ingeniería Informática
DIRECTORES DEL PROYECTO: Antonia Estero Botaro y Antonio García Domínguez.
AUTOR DEL PROYECTO: Valentín Liñeiro Barea.

Cádiz, 4 de febrero de 2014

Fdo.: Valentín Liñeiro Barea

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Este proyecto fin de carrera (PFC, de ahora en adelante), ha sido desarrollado por el alumno para colaborar en las investigaciones realizadas por el grupo de investigación "UCASE de Ingeniería del Software", en el seno de la Universidad de Cádiz. La actividad del grupo UCASE comprende actualmente las áreas de *Ingeniería de Servicios*, *Arquitecturas Dirigidas por Eventos*, *Arquitecturas Orientadas a Servicios*, *Desarrollo Dirigido por Modelos*, *Prueba de Software* y *Verificación y Validación de Software*.

Este PFC está orientado a la prueba de composiciones de servicios web (WS). El lenguaje empleado en las composiciones empleadas por el grupo de investigación es WS-BPEL 2.0 [37]. El lenguaje WS-BPEL tiene la ventaja de estar completamente expresado en XML, lo que hace que sea portable a cualquier motor estándar existente. Los WS permiten el desarrollo de aplicaciones distribuidas de manera rápida, simple y con coste bajo, motivo por el cual están cobrando protagonismo a la hora de definir procesos de negocio. Por lo tanto, la prueba de este tipo de software se torna esencial.

La técnica de prueba de software empleada por el grupo es la prueba de mutaciones, un tipo de prueba estructural basada en errores. En líneas generales, la técnica consiste en generar a partir de un programa una serie de programas, los cuales poseen únicamente una diferencia[1] respecto al original. Estos programas se denominan *mutantes*. Para generar los mutantes, se necesita una serie de reglas definidas, los *operadores de mutación*. Se suelen definir dos tipos de operadores de mutación, los que modelan fallos cometidos por los programadores y los que fuerzan una serie de criterios de cobertura.

El grupo de investigación UCASE ha desarrollado la herramienta MuBPEL [24]. Su trabajo consiste en el análisis de las composiciones WS-BPEL, la generación de los mutantes y su posterior ejecución. Incorpora un conjunto de operadores de mutación que tiene en cuenta los fallos

[1] Este tipo de mutación es de orden uno. Cabe la posibilidad de realizar más de un cambio sintáctico al programa original, obteniendo un mutante de orden superior.

sintácticos que pueden cometer los programadores [15] y un conjunto de operadores de mutación que permiten aplicar una serie de criterios de cobertura de código [14].

Por otra parte, el grupo de investigación UCASE ha desarrollado la herramienta TestGenerator [40]. Esta herramienta se encarga de la generación de conjuntos de casos de prueba aleatorios dada una especificación que sigue el formato TestSpec [47]. La salida generada por esta herramienta es utilizada por MuBPEL a la hora de la ejecución tanto del programa original como de los mutantes. El framework utilizado por MuBPEL en la ejecución de los casos de prueba es BPELUnit [31].

TestGenerator no tiene en cuenta el código del programa a la hora de obtener el conjunto de casos de prueba, lo que hace que generar casos de prueba que maten a mutantes concretos sea muy difícil. Con el objetivo de poder generar estos casos de prueba, existe una técnica, denominada *siembra automática* (o *Automatic Seeding*) [2, 3, 22], la cual tiene en cuenta las constantes del programa.

Este PFC se propuso para cubrir la necesidad del grupo UCASE de una herramienta que implemente la técnica de la siembra automática, dado que actualmente no dispone de ninguna. Esta herramienta, TestGenerator-Autoseed, estará basada en la existente, TestGenerator, y extenderá su funcionalidad implementando la técnica.

Una de las claves de este PFC es la escasez de bibliografía existente sobre el lenguaje WS-BPEL. Esto hace que el número de ejemplos de composiciones WS-BPEL existentes sea muy reducido. Esto convierte a este PFC en un proyecto de investigación, ya que será necesario un alto conocimiento del lenguaje para poder obtener ejemplos significativos para la aplicación de la técnica.

Este PFC también abarca el estudio comparativo entre la técnica de generación aleatoria de casos de prueba básica y la técnica de siembra automática.

1.1. Objetivos

Este PFC tiene como objetivo la creación de una herramienta de generación de casos de prueba que implemente la técnica de la siembra automática para composiciones WS-BPEL 2.0.

Para llevar a cabo el objetivo principal es necesario:

1. Adaptar la técnica de siembra automática al entorno donde se va a aplicar, es decir, las composiciones WS-BPEL 2.0, aplicando las optimizaciones que sean necesarias.
2. Implementar la herramienta que aplique la técnica, empleando como lenguaje de programación Java.
3. Crear un conjunto de casos de pruebas unitarias, empleando el framework JUnit [27]. Estos casos de prueba servirán para comprobar que la herramienta aplica correctamente la técnica a las composiciones WS-BPEL 2.0.

1.2. Alcance

Los productos generados por este PFC son los siguientes:

- La herramienta TestGenerator-Autoseed que implementa la técnica de generación de casos de prueba de siembra automática.
- Las clases Java que implementan las pruebas unitarias a la herramienta, empleando JUnit.
- Un estudio comparativo entre la técnica de generación de casos de prueba aleatoria y la técnica de siembra automática.

1.3. Visión general

El documento comienza con la definición de la técnica de la siembra automática. Más adelante, se comenta el calendario seguido para realizar este PFC y se realiza una breve descripción sobre las restricciones y tecnologías empleadas.

A continuación de lo anterior, se detalla el proceso de ingeniería seguido, es decir, las fases de análisis, diseño, implementación y pruebas.

El resto de la memoria se dedica a un estudio comparativo entre la técnica de generación aleatoria de casos de prueba y la técnica implementada en este PFC, además de las conclusiones extraídas y del trabajo futuro. Se adjuntan al final los manuales de usuario, desarrollador e instalación.

1.4. Pruebas de software

Las pruebas de software [42] consisten en verificar el comportamiento del software a partir de una serie de casos de prueba. Dado que la prueba de cada secuencia dentro del programa es imposible, ya que el número de posibles entradas es infinito, es necesario seleccionar dentro del dominio de entradas un subconjunto de ellas, las cuales conformarán el conjunto de casos de prueba.

Existen principalmente dos enfoques de prueba de software. Uno de ellos consiste en probar que el software funciona correctamente. Sin embargo, según Myers [36], la prueba de software consiste en *"el proceso de ejecutar un programa con la intención de encontrar errores"*. Esto aporta el segundo enfoque, probar que el software tiene errores. Este enfoque es el adecuado a la hora de realizar las pruebas, pues según Dijkstra [12], *"las pruebas sólo pueden demostrar la presencia de errores, no su ausencia"*.

Niveles de prueba de software

A continuación se presentan los principales niveles de prueba de software [42]. Los niveles se muestran en modo ascendente, es decir, desde los módulos hasta la prueba del sistema final.

Prueba de unidad Consiste en probar la lógica de cada módulo del software. Los resultados de las mismas pueden ser: *Éxito*, cuando se obtienen los resultados esperados; *Fracaso*, cuando los resultados obtenidos difieren de los esperados y *Error*, cuando los resultados obtenidos no se ajustan al dominio.

Prueba de integración Consiste en probar la integración de los módulos teniendo en cuenta la estructura del sistema. La integración puede ser:

- **Incremental** Las pruebas se realizan integrando los módulos uno a uno progresivamente.
- **No incremental** Las pruebas se realizan integrando todos los módulos a la vez.

Prueba de sistema Consiste en probar la integración entre software, hardware y usuario.

Prueba de aceptación Consiste en la prueba por parte del usuario en la que decide si el producto se ajusta a los requisitos especificados.

Tipos de prueba de software

Estáticas / dinámicas Las pruebas estáticas se realizan verificando el código del software, y las pruebas dinámicas se realizan ejecutando el software.

De caja blanca / negra Las pruebas de caja blanca están orientadas a la estructura interna del software, y las pruebas de caja negra están orientadas a la especificación del software, es decir, a sus posibles entradas y salidas.

1.4.1. Prueba de mutaciones

La prueba de mutaciones [25] es una técnica de prueba de software basada en errores. El criterio en que se basa esta técnica es el de medir la calidad de un conjunto de casos de prueba, lo que se denomina análisis de mutaciones. En la figura 1.1 extraída de [25], podemos ver el proceso de análisis de mutaciones.

Sea *P* el programa original. A partir de él, se genera un conjunto *P'* de programas, denominados *mutantes*. Los mutantes son programas que contienen una única diferencia respecto al programa original. Se generan aplicando una serie de reglas predefinidas, los *operadores de mutación*, al código fuente del programa original.

Los operadores de mutación introducen pequeños cambios sintácticos en el programa original, manteniendo la validez sintáctica del mismo, con el motivo de modelar los errores habituales cometidos por los programadores o forzar la aplicación de diferentes criterios de cobertura de código. Son dependientes del lenguaje de programación, así pues existen operadores de mutación para C [1], Ada [38], Fortran [28], Java [32], SQL [46], WS-BPEL [15], etc.

Un ejemplo de aplicación de un operador de mutación podría ser el siguiente:

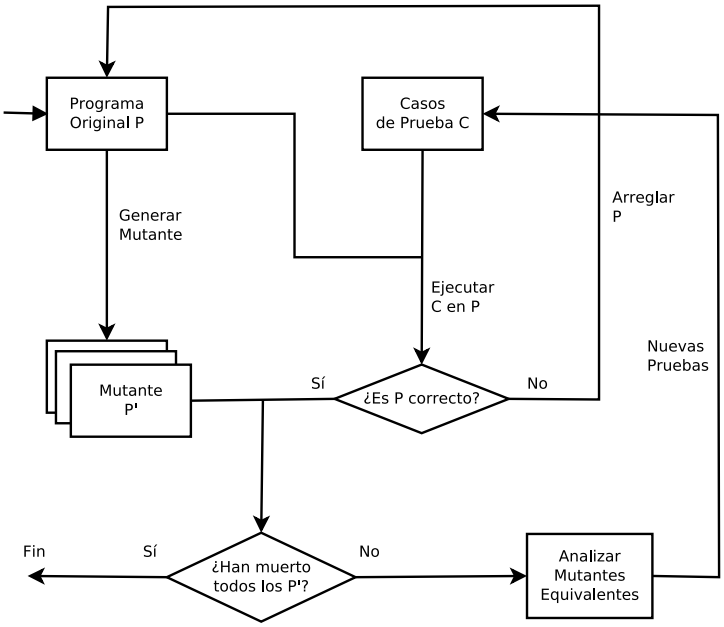


Figura 1.1.: Análisis de mutaciones

Programa original:

```
while(a - 2 > 0) {
      ...
}
```

Mutante generado tras aplicar el operador EIU, que añade el menos unario:

```
while(-(a - 2) > 0) {
      ...
}
```

El siguiente paso será ejecutar los mutantes sobre el conjunto de casos de prueba del programa original. En función de la salida del mutante, podemos diferenciar 3 estados:

- Si para todos los casos de prueba la salida del mutante coincide con la salida del programa original, el mutante está *vivo*.
- Si para algún caso de prueba la salida del mutante difiere de la salida del programa original, el mutante está *muerto*.
- Si el mutante generado no puede ser ejecutado, el mutante es *no válido*.

En algunos casos, algunos mutantes vivos siempre poseen la misma salida que el programa original, debido al cambio aplicado. Éstos se denominan mutantes equivalentes. Así pues, tenemos los siguientes tipos de mutantes:

No válido Mutante que no puede ser ejecutado.

Muerto Mutante cuya salida difiere de la del programa original para alguno de los casos de prueba.

Vivo Mutante cuya salida coincide con la del programa original para todos los casos de prueba disponibles.

- **Persistente** No existen suficientes casos de prueba para matarlo.
- **Equivalente** Provoca siempre la misma salida que el programa original.

Para medir la calidad de un conjunto de casos de prueba es necesario calcular la puntuación de mutación, que consiste en el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes.

El problema que posee el empleo de esta técnica de prueba del software es el coste computacional que conlleva el proceso, ya que existen multitud de operadores de mutación que, aplicados al programa original, generan una gran cantidad de mutantes, los cuales deben ser ejecutados hasta encontrar un caso de prueba que los mate. Por lo tanto, existen una serie de técnicas que ayudan a reducir el coste computacional, como las siguientes:

Técnicas para reducir el número de mutantes

Mutación aleatoria Consiste en ejecutar un subconjunto de mutantes seleccionados aleatoriamente.

Agrupación de mutantes Consiste en agrupar los mutantes en torno a los casos de prueba que los matan, y seleccionar algunos mutantes de cada grupo.

Mutación selectiva Consiste en aplicar un subconjunto de los operadores de mutación disponibles.

Mutación evolutiva Consiste en la ejecución de un subconjunto de mutantes seleccionados mediante un algoritmo genético.

Mutación de orden superior Consiste en aplicar más de un operador de mutación por cada mutante.

Técnicas para reducir el coste de la ejecución de los mutantes

Mutación fuerte Se compara la salida del mutante con la del programa original tras finalizar la ejecución.

Mutación débil Se compara el estado del mutante una vez ejecutado el punto donde reside la mutación.

Optimización del tiempo de ejecución Son técnicas que se apoyan en la optimización aportada por compiladores, intérpretes o metamutantes.

Empleo de plataformas avanzadas Se basan en la aplicación del paralelismo a la hora de ejecutar los mutantes en un cluster, equipos multiprocesadores, redes de ordenadores, etc.

1.4.2. Generación aleatoria de casos de prueba

La generación aleatoria de casos de prueba [45] es un método de automatización de la generación de casos de prueba consistente en obtener un conjunto de casos de prueba con valores arbitrarios pertenecientes al dominio de la entrada.

Este método se encuadra dentro de los denominados métodos basados en la interfaz, ya que desconoce tanto la especificación del programa como su implementación. Sus características son:

- Eficiencia. La generación aleatoria es eficiente, ya que únicamente necesita un generador de números aleatorios unido a cierta lógica que permita transformar los valores obtenidos.
- Ineficacia. La generación aleatoria es ineficaz, debido a que genera conjuntos de casos de prueba de escasa calidad.

La generación aleatoria tiene como principales aplicaciones la prueba de software muy complejo y específico y, sobretodo, servir de método base para comparar la eficacia de otros métodos de generación de casos de prueba.

1.5. El lenguaje WS-BPEL 2.0

El lenguaje WS-BPEL 2.0 es un lenguaje estandarizado por OASIS [37], que permite definir procesos de negocio a partir de la composición de WS. Está completamente basado en XML [7], cualidad que lo hace portable e independiente del motor donde se ejecute.

Un proceso WS-BPEL se divide en las siguientes secciones:

1. Definición de relaciones con los socios externos, es decir, el cliente y los WS involucrados en el proceso.
2. Definición de las variables empleadas por el proceso, empleando las tecnologías WSDL [9] y XML Schema (XSD) [5].
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso:

   Manejadores de fallo Contienen la lógica necesaria para gestionar los fallos ocurridos durante la ejecución del proceso o la de un WS involucrado.

   Manejadores de eventos Contienen la acciones necesarias a la hora de recibir peticiones durante la ejecución del proceso.

   Manejadores de terminación Indican las acciones necesarias para terminar el proceso ordenadamente.

   Manejadores de compensación Deshacen la invocación de un WS.

4. Descripción del proceso de negocio mediante las actividades que proporciona el lenguaje.

Los elementos anteriores son todos globales por defecto. Cabe la posibilidad de declarar estos elementos de forma local, empleando el contenedor `scope`. A través de este elemento, se puede dividir el proceso de negocio en distintos ámbitos.

Un proceso WS-BPEL está formado por actividades. Una actividad está representada por un elemento XML, y se le pueden asociar una serie de atributos y un conjunto de contenedores, los cuales pueden poseer atributos asociados. Las actividades pueden ser de los siguientes tipos:

Básicas Son actividades que tienen una labor determinada dentro del proceso de negocio (recepción de mensajes, invocación de servicios, respuesta al cliente, etc.)

Estructuradas Son actividades compuestas por actividades básicas, y son las encargadas de definir la lógica de negocio.

El lenguaje WS-BPEL tiene soporte nativo para la concurrencia, mediante la actividad `flow`. Esta actividad permite la ejecución concurrente de un conjunto de actividades, indicando las condiciones de sincronización necesarias. Un ejemplo de esta actividad es el siguiente:

```
<flow> ← Actividad estructurada
 <links> ← Contenedor
  <link name="comprobarVuelo-reservarVuelo" ← Atributo/>
 </links>
 <invoke name="comprobarVuelo" ... > ← Actividad básica
  <sources> ← Contenedor
   <source linkName="comprobarVuelo-reservarVuelo"/> ← Elemento
  </sources>
 </invoke>
 <invoke name="comprobarHotel" ... />
 <invoke name="comprobarAlquilerCoche" ... />
 <invoke name="reservarVuelo" ... >
  <targets> ← Contenedor
   <target linkName="comprobarVuelo-reservarVuelo"/> ← Elemento
  </targets>
 </invoke>
</flow>
```

Se puede apreciar que la actividad `flow` invoca a tres WS de manera concurrente, `comprobarVuelo`, `comprobarHotel` y `comprobarAlquilerCoche`. El WS `reservarVuelo` sólo se invocará si termina el servicio comprobarVuelo. Tenemos un ejemplo de sincronización de actividades, pues se ha establecido un enlace (`link`) entre ellas. De esta manera, la actividad objetivo del enlace se ejecuta sólo si la actividad fuente se completa.

WS-BPEL soporta distintos lenguajes de expresiones. Todos los motores WS-BPEL estándar existentes soportan el lenguaje de expresiones XPath 1.1 [4]. EL lenguaje XPath es un lenguaje declarativo que permite la realización de consultas en documentos XML. XPath dispone de operadores

aritméticos, lógicos y relacionales con una sintaxis muy similar a la de los lenguajes tradicionales.

1.6. Glosario

1.6.1. Acrónimos

CFG Control Flow Graph

CLI Command Line Interface

GUI Graphical User Interface

IDE Integrated Development Environment

SCV Sistema de Control de Versiones

SO Sistema Operativo

SOAP Simple Object Access Protocol

UML Unified Modeling Language

VTL Velocity Template Language

WS Web Services

WSDL Web Services Description Language

WS-BPEL Web Services Business Process Execution Language

XML eXtensible Markup Language

XPath XML Path Language

XSD XML Schema Definition

YAML Yet Another Markup Language

1.6.2. Definiciones

BPR Extensión del fichero que contiene todos los elementos necesarios para el despliegue de una composición WS-BPEL.

Desplegar En el mundo de los WS, desplegar un proceso consiste en ponerlo en funcionamiento en un motor, donde queda a la espera de las peticiones de los clientes en un puerto especificado.

Mockup Sustituto de un servicio web real empleado a la hora de probar la ejecución de composiciones WS-BPEL con BPELUnit. Se emplea para evitar la utilización de servicios web reales, por motivos de precio o de control del proceso de prueba. Se encargan de responder empleando valores predefinidos, especificados previamente.

PDD Extensión del fichero empleado por el motor ActiveBPEL que contiene la información necesaria para el despliegue de un proceso WS-BPEL.

2. Generación de casos de prueba mediante siembra automática

En este capítulo se definirá la técnica de siembra automática a nivel teórico, y se detallará su aplicación a composiciones WS-BPEL 2.0.

2.1. Definición

La siembra automática [2, 3, 22] es una técnica de automatización de la generación de casos de prueba que se basa en extender la técnica de generación aleatoria utilizando las constantes existentes dentro del programa a probar.

Esta técnica puede encuadrarse [45] dentro de las técnicas de automatización mixtas, ya que combina una técnica de automatización basada en la interfaz, que es la técnica aleatoria, con el uso de las constantes existentes en el programa, lo que la encuadra dentro de las técnicas basadas en la implementación.

2.1.1. Generación de casos de prueba

A continuación veremos cómo se generan los casos de prueba aplicando la siembra automática. En primer lugar veremos los pasos necesarios para aplicar la técnica, que son los siguientes:

1. Obtener las constantes del programa.
2. Obtener las variables que forman parte de la especificación de los casos de prueba.
3. Asignar a cada variable las constantes que son compatibles con ella en el sentido de los tipos de datos.

4. Generar un conjunto de casos de prueba aleatorio inicial.
5. Por cada caso de prueba, realizar una modificación sustituyendo en la variable seleccionada el valor de constante elegido.

Una vez realizados estos pasos, obtenemos un conjunto de casos de prueba. Supongamos que tenemos una colección de tipos básicos T en el lenguaje en el que están expresados los programas donde se aplique esta técnica. Entonces, tendremos, para cada tipo $i \in T$, una colección V_i de variables del tipo y una colección C_i de constantes compatibles con el tipo, y por tanto, con el conjunto V_i.

El número de casos de prueba que se generan, p, se puede obtener entonces a partir de la siguiente fórmula:

$$\sum_{i=1}^{|T|} |V_i| \cdot |C_i|$$

Dado t, índice dentro de la colección de un tipo cualquiera que pertenece a T, i, índice dentro de la colección de una variable que pertenece a V_t y j, índice dentro de la colección de una constante que pertenece a C_t, podremos localizar el caso de prueba donde se produce el cambio de la siguiente forma:

$$c_k : V_{ti} \leftarrow C_{tj},\ k = |T| \cdot (t-1) + |V_t| \cdot (i-1) + j$$

2.1.1.1. Ejemplo

En el listado 2.1 tenemos un fragmento de código al que podemos aplicar la siembra automática para obtener un conjunto de casos de prueba.

En este caso, tendríamos lo siguiente:

- Colección de tipos: $T = \{string, int\}$. En la fórmula para calcular NCP, se indexan por el orden establecido ($string = 1$, $int = 2$).
- Colecciones de variables por tipo: $V_1 = \{estado\}$ y $V_2 = \{temperatura, presion\}$.
- Colecciones de constantes por tipo: $C_1 = \{ABIERTO, CERRADO\}$ y $C_2 = \{40, 400\}$.

Listado 2.1: Fragmento de ejemplo para la aplicación de la siembra automática

```
if(estado == "ABIERTO") {
   while(temperatura > 40 && presion > 400) {
      ...
   }
   estado = "CERRADO";
}
```

Dado que $|T| = 2$, $|V_1| = 1$, $|V_2| = 2$, $|C_1| = 2$ y $|C_2| = 2$, el número de casos de prueba sería:

$$p = \sum_{i=1}^{|T|} |V_i| \cdot |C_i| = \sum_{i=1}^{2} |V_i| \cdot |C_i| = 1 \cdot 2 + 2 \cdot 2 = 6$$

Por lo tanto, necesitamos generar 6 casos de prueba para aplicar la técnica. Cada cambio se vería reflejado de la siguiente forma:

- $c_{2\cdot0+1\cdot0+1} : V_{11} \leftarrow C_{11} \rightarrow c_1 :$ `estado` $\leftarrow$ `ABIERTO`.
- $c_{2\cdot0+1\cdot0+2} : V_{11} \leftarrow C_{12} \rightarrow c_2 :$ `estado` $\leftarrow$ `CERRADO`.
- $c_{2\cdot1+2\cdot0+1} : V_{21} \leftarrow C_{21} \rightarrow c_3 :$ `temperatura` $\leftarrow$ `40`.
- $c_{2\cdot1+2\cdot0+2} : V_{21} \leftarrow C_{22} \rightarrow c_4 :$ `temperatura` $\leftarrow$ `400`.
- $c_{2\cdot1+2\cdot1+1} : V_{22} \leftarrow C_{21} \rightarrow c_5 :$ `presion` $\leftarrow$ `40`.
- $c_{2\cdot1+2\cdot1+2} : V_{22} \leftarrow C_{22} \rightarrow c_6 :$ `presion` $\leftarrow$ `400`.

En la tabla 2.1 podemos ver el conjunto de casos de prueba generado. Los valores de las variables con puntos suspensivos representan valores aleatorios cualesquiera dentro del dominio de cada variable.

2.1.2. Optimización

Dependiendo del programa para el que estemos generando el conjunto de casos de prueba, puede haber muchas variables a las que se le pueda asignar un conjunto de constantes compatibles de gran tamaño. Esto quiere decir que $|V_i|$ y $|C_i|$ serán grandes, y por lo tanto, p también.

Casos de prueba						
	1	2	3	4	5	6
estado	ABIERTO	CERRADO	...	...	...	...
temperatura	...	...	40	400	...	...
presion	...	...	...	...	40	400

Tabla 2.1.: Conjunto de casos de prueba generado para el fragmento del listado 2.1.

Este hecho, en ciertos entornos donde la ejecución de cada caso de prueba sea costosa en términos de tiempo o espacio, supone un problema para aplicar la siembra automática. Para paliar este problema, surge una optimización de la técnica, la cual se basa en ampliar la restricción de compatibilidad de las variables y las constantes.

Para cada tipo $t \in T$, cada variable $v_t \in V_t$ se relaciona con todas las constantes que pertenecen a C_t. De esta forma, generamos gran cantidad de casos de prueba que no son significativos, pues únicamente nos interesan aquellos donde las variables toman las constantes con los que están relacionadas, sea por comparación o por asignación, dentro del programa.

Para resolver este problema, planteamos una optimización del método inicial la cual consiste en obligar a que la variable esté relacionada con la constante dentro del programa para ser compatible con ella. Con esta nueva restricción, obtenemos:

$$p' = \sum_{i=1}^{|T|} \sum_{j=1}^{|V_i|} |C_{ij}|$$

En esta expresión aparece $|C_{ij}|$, que es el cardinal de la colección de constantes de un tipo dado i compatibles y relacionadas con una variable j. Dado que $|C_{ij}| \leq |C_i|$, entonces tendremos que $p' \leq p$.

La diferencia entre p' y p depende obviamente del programa donde se aplique la técnica. La eficacia de la optimización puede medirse, para cada tipo i, como $|C_i| - |C_{ij}|$. A mayor valor, mayor eficacia de la optimización. Cuando aplicamos la optimización, la forma de localizar los casos

de prueba difiere, siendo:

$$c_k : V_{ti} \leftarrow C_{ij},\ k = |T| \cdot (t-1) + |C_{ij}| \cdot (i-1) + j$$

2.1.2.1. Ejemplo

Utilizaremos de nuevo el fragmento del listado 2.1 para generar el conjunto de casos de prueba, en este caso aplicando la optimización definida.

Para este ejemplo, las colecciones T, V_1 y V_2 son los mismos que en el ejemplo 2.1.1.1. Sin embargo, debido a la optimización, las colecciones C_1 y C_2 no se toman al completo, si no las subcolecciones de constantes que se relacionan con cada variable: $C_{11} = \{ABIERTO, CERRADO\}$, $C_{21} = \{40\}$ y $C_{22} = \{400\}$.

Sabiendo que $|T| = 2$, $|V_1| = 1$, $|V_2| = 2$, $|C_{11}| = 2$, $|C_{21}| = 1$ y $|C_{22}| = 1$, el número de casos de prueba sería:

$$\begin{aligned} p' &= \sum_{i=1}^{|T|} \sum_{j=1}^{|V_i|} |C_{ij}| = \sum_{i=1}^{2} \sum_{j=0}^{|V_i|} |C_{ij}| \\ &= \sum_{j=1}^{1} |C_{1j}| + \sum_{j=1}^{2} |C_{2j}| = 2 + 1 + 1 = 4 \end{aligned}$$

Como podemos ver, en este caso sólo necesitamos generar 4 casos de prueba frente a los 6 que generábamos sin aplicar la optimización. Los cambios a realizar serían:

- $c_{2\cdot0+2\cdot0+1} : V_{11} \leftarrow C_{11} \rightarrow c_1 : \texttt{estado} \leftarrow \texttt{ABIERTO}$.
- $c_{2\cdot0+2\cdot0+2} : V_{11} \leftarrow C_{12} \rightarrow c_2 : \texttt{estado} \leftarrow \texttt{CERRADO}$.
- $c_{2\cdot1+1\cdot0+1} : V_{21} \leftarrow C_{21} \rightarrow c_3 : \texttt{temperatura} \leftarrow \texttt{40}$.
- $c_{2\cdot1+1\cdot1+1} : V_{22} \leftarrow C_{22} \rightarrow c_4 : \texttt{presion} \leftarrow \texttt{400}$.

En la tabla 2.2 está el conjunto de casos de prueba generado. Se puede comprobar que respecto a la tabla 2.1 desaparecen los casos de prueba 4 y 5 de la misma, los que no son significativos.

Casos de prueba				
	1	2	3	4
estado	ABIERTO	CERRADO	...	...
temperatura	...	...	40	...
presion	...	...	...	400

Tabla 2.2.: Conjunto de casos de prueba generado para el fragmento del listado 2.1 aplicando optimización.

2.2. Aplicación de la técnica a composiciones WS-BPEL 2.0

En la sección 2.1.1 se definió a nivel teórico el procedimiento necesario para aplicar la siembra automática a un programa y obtener así un conjunto de casos de prueba inicial.

En el caso del lenguaje WS-BPEL, las diferentes herramientas utilizadas a la hora de ejecutar las composiciones influyen en la aplicación de la técnica.

La ejecución de una composición empleando MuBPEL requiere de un conjunto de casos de prueba expresados en un fichero `.bpts`, empleando el framework BPELUnit. Este fichero permite la generación de casos de prueba empleando plantillas parametrizables utilizando el lenguaje de plantillas Apache Velocity [18].

Las variables empleadas en las plantillas de los casos de prueba se definen empleando un lenguaje denominado TestSpec [47]. Mediante una especificación en este lenguaje (fichero `.spec`) e indicando el número de casos de prueba que se desean, la herramienta TestGenerator obtiene un fichero en lenguaje Apache Velocity con valores aleatorios para las variables de la especificación por cada caso de prueba.

Una vez conocido el entorno, podemos aplicar la técnica de siembra automática de acuerdo a los siguientes pasos:

1. Se obtienen las constantes existentes en la composición WS-BPEL a probar, junto a las variables de la composición con las que están relacionadas.

2. Se obtienen las variables de la especificación.
3. Se hallan las relaciones existentes entre las variables de la especificación y las variables de la composición.
4. Se determina qué constantes pueden asignarse a las variables de la especificación, teniendo en cuenta las restricciones de las mismas.
5. Se calcula el número de casos de prueba necesarios mediante la fórmula de la sección 2.1.1.
6. Se genera mediante TestGenerator un conjunto de casos de prueba con valores aleatorios base, del tamaño calculado en el paso anterior.
7. Siguiendo el estilo de localización propuesto, se modifican los casos de prueba sustituyendo el valor aleatorio en la variable correspondiente por la constante asociada.
8. Se vuelca el conjunto a un fichero en formato Apache Velocity (`data.vm`).

El tercer paso correspondería a la aplicación de la optimización. Se detallará adecuadamente en la subsección siguiente.

Una vez concluido el proceso, se obtiene un fichero con los datos necesarios para la plantilla de BPELUnit, los cuales pueden utilizarse en MuBPEL a la hora de ejecutar la composición.

2.2.1. Aplicación de la optimización

En la sección 2.2 se detalló el proceso necesario para generar un conjunto de casos de prueba mediante siembra automática. Uno de los pasos de ese proceso, el de la búsqueda de las relaciones existentes entre las variables de la especificación en TestSpec y las variables de la composición WS-BPEL corresponde a la aplicación de la optimización que se propone a nivel teórico.

En el lenguaje WS-BPEL [37], las variables pueden definirse indicando un tipo de mensaje de WSDL [9] o un elemento o tipo de XML Schema [5]. Con el objetivo de unificar el tratamiento de las variables y obtener de manera simple la estructura contenida en las mismas, el grupo UCASE desarrolló la herramienta ServiceAnalyzer [26]. Esta herramienta, empleando como entrada las ficheros `.wsdl` pertenecientes a los partners

que participan en la composición, es capaz de generar un catálogo en XML, el cual contiene para cada variable la estructura interna de la misma.

Una vez conocida la estructura, es necesario acceder al conjunto de casos de prueba en BPELUnit, el cual sirve de enlace entre las variables de la especificación y las de la composición, ya que contiene las plantillas en las que se detalla de forma explícita la forma en la que se utilizan dentro de la estructura de las variables de la composición las variables declaradas dentro de la especificación.

Una vez detallada la mecánica, el proceso podríamos describirlo a través de los siguientes pasos:

1. Se crea mediante ServiceAnalyzer el catálogo que contiene la estructura interna de las variables de la composición.
2. Se obtienen las plantillas de las variables en el conjunto de casos de prueba de la composición (fichero `.bpts`).
3. Se comparan las plantillas del catálogo con las obtenidas en el paso anterior, estableciendo igualdades.
4. Se busca en el interior de las plantillas coincidentes extraídas del fichero `.bpts` las variables de la especificación.
5. Si el contenido coincide con alguna de ellas, se establece la relación entre la variable de la especificación y la variable de la composición.
6. Si no se encuentran coincidencias, se busca el origen de la variable y se repite el proceso.

No puede culminar el proceso si no encontramos una coincidencia de plantillas de forma inmediata, ya que en la mayoría de composiciones las constantes se relacionan con variables de la composición que no están directamente ligadas con las variables de la especificación, caso de variables temporales, por ejemplo. En este caso, es necesario obtener el origen de estas variables, que es otra variable de la composición la cual sí está relacionada con una variable de la especificación.

El cálculo del origen se realiza aplicando un algoritmo de análisis de flujo de datos a la composición dada. Debido a que en la actualidad no existe ninguna herramienta dentro del grupo que se encargue de este trabajo y a que desarrollarla sería un trabajo correspondiente a un proyecto distinto, el cálculo del origen se realiza actualmente buscando el origen de

la cadena de asignaciones donde participe la variable cuya relación queremos obtener. Este método sencillo permite obtener algunas relaciones que no se encontraban con anterioridad.

3. Calendario

En este capítulo se detalla el calendario seguido para llevar a cabo este PFC.

3.1. Fases del proyecto

A continuación se detallan las diferentes pases de las que se ha compuesto este PFC. Las fechas concretas y duraciones de las mismas pueden consultarse en el diagrama de Gantt presente en las figuras 3.1, 3.2 y 3.3.

Primera fase

En esta fase, se tomaron los requisitos iniciales de la herramienta TestGenerator-Autoseed, plasmándolos en su primera versión.

El grupo indicó la necesidad de implementar una herramienta que permitiese generar un conjunto de casos de prueba aplicando la técnica de siembra automática, con el objetivo de poder realizar estudios específicos para comparar la efectividad de esta técnica respecto a la generación aleatoria en las composiciones WS-BPEL y mejorar sus conjuntos de casos de prueba.

Dado que el alumno ya conocía las tecnologías necesarias para la realización de este PFC (Ver 4.4.5) dado que fueron utilizadas en su PFC de la Ingeniería Técnica en Informática de Sistemas [29], se pasó directamente a desarrollar la primera versión de la herramienta, con el objetivo de tomar los requisitos iniciales de la misma.

Esta versión sólo era funcional para la composición `LoanApprovalDoc` y tenía en cuenta únicamente las constantes y variables de tipo `string` e `int`.

Segunda fase

En esta fase se generalizó la herramienta, de manera que funcionase para todas las composiciones WS-BPEL existentes en el repositorio del grupo UCASE.

En la segunda versión además se refactorizó el código fuente de la misma, con el objetivo de facilitar la inclusión de nueva funcionalidad de manera cómoda.

Tercera fase

En esta fase se extendió la herramienta para que aplicase la siembra automática teniendo en cuenta todos los tipos básicos existentes. Para ello, se añadió el soporte para variables y constantes de tipo `float`, `date`, `time`, `dateTime` y `duration` en la tercera versión de la misma.

Cabe destacar además que en esta fase se realizó una modularización de la herramienta, con el objetivo de permitir la aplicación de la técnica a otros entornos y lenguajes de programación en el futuro.

Cuarta fase

En esta fase se completó la definición de la técnica teniendo en cuenta los tipos compuestos existentes, tuplas y listas. Una vez definido el comportamiento ante estos tipos de datos, se les dió soporte en la cuarta versión de la herramienta.

Adicionalmente, se dotó a la herramienta de capacidad para generar un registro de variables y constantes compatibles con las mismas, clasificadas por el tipo.

En esta fase comenzó la redacción de la documentación de este PFC, la cual se ha realizado de forma paralela a las fases restantes del mismo.

Quinta fase

En esta fase se definió la optimización de la técnica, tras observar el elevado número de casos de prueba generados para las composiciones WS-BPEL más complejas del repositorio del grupo con la técnica original.

Una vez definida la optimización a realizar, se procedió a integrarla dentro de la herramienta en su quinta versión. Dada la complejidad del proceso de aplicación de la optimización en este entorno, ésta puede considerarse la fase más importante del proyecto.

Una vez realizada esta tarea, se enriqueció la información del informe que genera TestGenerator-Autoseed, permitiendo obtener información relativa al proceso de optimización. Se añadió la posibilidad de graduar el nivel de detalle del informe, y el formato de salida del mismo, añadiendo la posibilidad de exportar el informe a XML. Toda la nueva funcionalidad relativa al informe también se añadió en la quinta versión.

Última fase

Esta fase se inició con la realización de un estudio comparativo entre la técnica aleatoria y la técnica de siembra automática (ver 6).

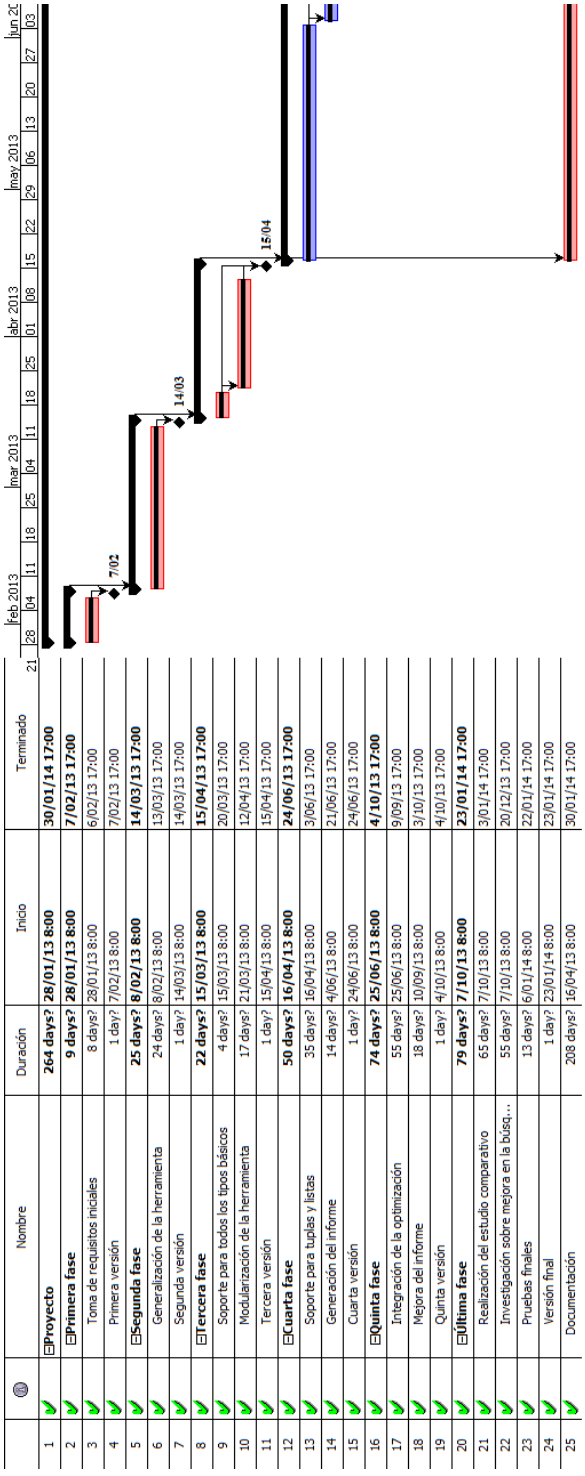
Dados los resultados del estudio, en los que se aprecia que la optimización no es del todo efectiva para algunas composiciones WS-BPEL del repositorio, se decidió dedicar un período de investigación para intentar mejorar la búsqueda del origen de las variables dentro de la composición.

Como conclusión se extrajo que, para mejorar los resultados de la optimización en este tipo de composiciones era necesaria la aplicación de un algoritmo de análisis de flujo de datos. Debido a que los trabajos existentes en torno a la temática anterior para WS-BPEL no cumplen los requisitos para poder aplicar el algoritmo a corto plazo, bien por falta de madurez o bien por falta de la existencia de una estructura necesaria para aplicarlos (CFG para WS-BPEL), se decidió implantar una mejora basada en la búsqueda entre asignaciones, quedando como trabajo futuro la aplicación del algoritmo de análisis de flujo de datos.

Finalmente, se completó la documentación del PFC y se añadió la mejora explicada anteriormente a la herramienta en su versión final, realizando además las pruebas finales a la misma.

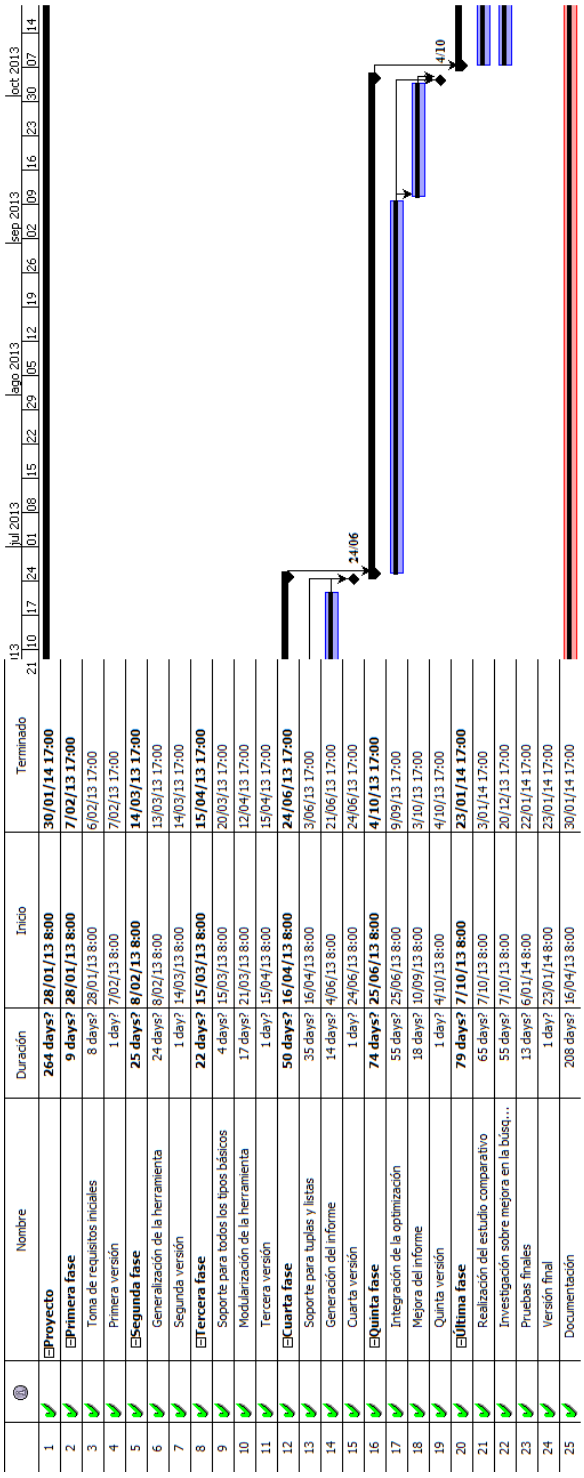
3.2. Gestión del tiempo y recursos

En la figuras 3.1, 3.2 y 3.3 se presenta el diagrama de Gantt que refleja la gestión del tiempo empleado para realizar este PFC. Se pueden distinguir claramente las fases del proyecto, representadas como tareas, y las subtareas que las conforman. Se puede apreciar gráficamente la concurrencia entre algunas tareas, por su similitud, como se especificó con anterioridad.



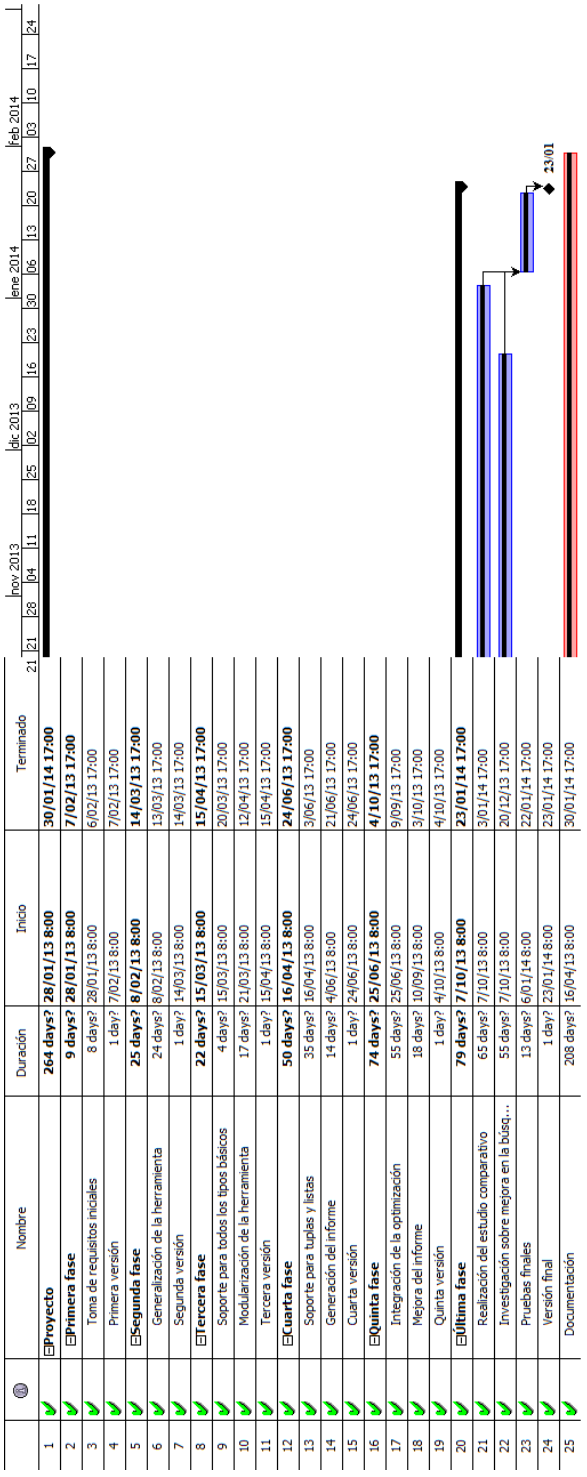
	Nombre	Duración	Inicio	Terminado
1	**Proyecto**	**264 days?**	**28/01/13 8:00**	**30/01/14 17:00**
2	**Primera fase**	**9 days?**	**28/01/13 8:00**	**7/02/13 17:00**
3	Toma de requisitos iniciales	8 days?	28/01/13 8:00	6/02/13 17:00
4	Primera versión	1 day?	7/02/13 8:00	7/02/13 17:00
5	**Segunda fase**	**25 days?**	**8/02/13 8:00**	**14/03/13 17:00**
6	Generalización de la herramienta	24 days?	8/02/13 8:00	13/03/13 17:00
7	Segunda versión	1 day?	14/03/13 8:00	14/03/13 17:00
8	**Tercera fase**	**22 days?**	**15/03/13 8:00**	**15/04/13 17:00**
9	Soporte para todos los tipos básicos	4 days?	15/03/13 8:00	20/03/13 17:00
10	Modularización de la herramienta	17 days?	21/03/13 8:00	12/04/13 17:00
11	Tercera versión	1 day?	15/04/13 8:00	15/04/13 17:00
12	**Cuarta fase**	**50 days?**	**16/04/13 8:00**	**24/06/13 17:00**
13	Soporte para tuplas y listas	35 days?	16/04/13 8:00	3/06/13 17:00
14	Generación del informe	14 days?	4/06/13 8:00	21/06/13 17:00
15	Cuarta versión	1 day?	24/06/13 8:00	24/06/13 17:00
16	**Quinta fase**	**74 days?**	**25/06/13 8:00**	**4/10/13 17:00**
17	Integración de la optimización	55 days?	25/06/13 8:00	9/09/13 17:00
18	Mejora del informe	18 days?	10/09/13 8:00	3/10/13 17:00
19	Quinta versión	1 day?	4/10/13 8:00	4/10/13 17:00
20	**Última fase**	**79 days?**	**7/10/13 8:00**	**23/01/14 17:00**
21	Realización del estudio comparativo	65 days?	7/10/13 8:00	3/01/14 17:00
22	Investigación sobre mejora en la búsq...	55 days?	7/10/13 8:00	20/12/13 17:00
23	Pruebas finales	13 days?	6/01/14 8:00	22/01/14 17:00
24	Versión final	1 day?	23/01/14 8:00	23/01/14 17:00
25	Documentación	208 days?	16/04/13 8:00	30/01/14 17:00

Figura 3.1.: Diagrama de Gantt, primera parte



		Nombre	Duración	Inicio	Terminado
1	✓	**Proyecto**	**264 days?**	**28/01/13 8:00**	**30/01/14 17:00**
2	✓	**Primera fase**	**9 days?**	**28/01/13 8:00**	**7/02/13 17:00**
3	✓	Toma de requisitos iniciales	8 days?	28/01/13 8:00	6/02/13 17:00
4	✓	Primera versión	1 day?	7/02/13 8:00	7/02/13 17:00
5	✓	**Segunda fase**	**25 days?**	**8/02/13 8:00**	**14/03/13 17:00**
6	✓	Generalización de la herramienta	24 days?	8/02/13 8:00	13/03/13 17:00
7	✓	Segunda versión	1 day?	14/03/13 8:00	14/03/13 17:00
8	✓	**Tercera fase**	**22 days?**	**15/03/13 8:00**	**15/04/13 17:00**
9	✓	Soporte para todos los tipos básicos	4 days?	15/03/13 8:00	20/03/13 17:00
10	✓	Modularización de la herramienta	17 days?	21/03/13 8:00	12/04/13 17:00
11	✓	Tercera versión	1 day?	15/04/13 8:00	15/04/13 17:00
12	✓	**Cuarta fase**	**50 days?**	**16/04/13 8:00**	**24/06/13 17:00**
13	✓	Soporte para tuplas y listas	35 days?	16/04/13 8:00	3/06/13 17:00
14	✓	Generación del informe	14 days?	4/06/13 8:00	21/06/13 17:00
15	✓	Cuarta versión	1 day?	24/06/13 8:00	24/06/13 17:00
16	✓	**Quinta fase**	**74 days?**	**25/06/13 8:00**	**4/10/13 17:00**
17	✓	Integración de la optimización	55 days?	25/06/13 8:00	9/09/13 17:00
18	✓	Mejora del informe	18 days?	10/09/13 8:00	3/10/13 17:00
19	✓	Quinta versión	1 day?	4/10/13 8:00	4/10/13 17:00
20	✓	**Ultima fase**	**79 days?**	**7/10/13 8:00**	**23/01/14 17:00**
21	✓	Realización del estudio comparativo	65 days?	7/10/13 8:00	3/01/14 17:00
22	✓	Investigación sobre mejora en la búsq...	55 days?	7/10/13 8:00	20/12/13 17:00
23	✓	Pruebas finales	13 days?	6/01/14 8:00	22/01/14 17:00
24	✓	Versión final	1 day?	23/01/14 8:00	23/01/14 17:00
25	✓	Documentación	208 days?	16/04/13 8:00	30/01/14 17:00

Figura 3.2.: Diagrama de Gantt, segunda parte



		Nombre	Duración	Inicio	Terminado
1	✓	**Proyecto**	**264 days?**	**28/01/13 8:00**	**30/01/14 17:00**
2	✓	**Primera fase**	**9 days?**	**28/01/13 8:00**	**7/02/13 17:00**
3	✓	Toma de requisitos iniciales	8 days?	28/01/13 8:00	6/02/13 17:00
4	✓	Primera versión	1 day?	7/02/13 8:00	7/02/13 17:00
5	✓	**Segunda fase**	**25 days?**	**8/02/13 8:00**	**14/03/13 17:00**
6	✓	Generalización de la herramienta	24 days?	8/02/13 8:00	13/03/13 17:00
7	✓	Segunda versión	1 day?	14/03/13 8:00	14/03/13 17:00
8	✓	**Tercera fase**	**22 days?**	**15/03/13 8:00**	**15/04/13 17:00**
9	✓	Soporte para todos los tipos básicos	4 days?	15/03/13 8:00	20/03/13 17:00
10	✓	Modularización de la herramienta	17 days?	21/03/13 8:00	12/04/13 17:00
11	✓	Tercera versión	1 day?	15/04/13 8:00	15/04/13 17:00
12	✓	**Cuarta fase**	**50 days?**	**16/04/13 8:00**	**24/06/13 17:00**
13	✓	Soporte para tuplas y listas	35 days?	16/04/13 8:00	3/06/13 17:00
14	✓	Generación del informe	14 days?	4/06/13 8:00	21/06/13 17:00
15	✓	Cuarta versión	1 day?	24/06/13 8:00	24/06/13 17:00
16	✓	**Quinta fase**	**74 days?**	**25/06/13 8:00**	**4/10/13 17:00**
17	✓	Integración de la optimización	55 days?	25/06/13 8:00	9/09/13 17:00
18	✓	Mejora del informe	18 days?	10/09/13 8:00	3/10/13 17:00
19	✓	Quinta versión	1 day?	4/10/13 8:00	4/10/13 17:00
20	✓	**Última fase**	**79 days?**	**7/10/13 8:00**	**23/01/14 17:00**
21	✓	Realización del estudio comparativo	65 days?	7/10/13 8:00	3/01/14 17:00
22	✓	Investigación sobre mejora en la búsq...	55 days?	7/10/13 8:00	20/12/13 17:00
23	✓	Pruebas finales	13 days?	6/01/14 8:00	22/01/14 17:00
24	✓	Versión final	1 day?	23/01/14 8:00	23/01/14 17:00
25	✓	Documentación	208 days?	16/04/13 8:00	30/01/14 17:00

Figura 3.3.: Diagrama de Gantt, tercera parte

4. Descripción general del proyecto

En este capítulo se realizará una descripción general de este PFC.

4.1. Perspectiva del producto

4.1.1. Entorno del producto

La herramienta **TestGenerator-Autoseed** utiliza como base a TestGenerator [40].

TestGenerator es una herramienta orientada a la generación aleatoria de casos de prueba, de forma que mediante una especificación en formato TestSpec [47], genera un conjunto de casos de prueba con valores aleatorios del tamaño que se le haya especificado. TestGenerator está formada por las siguientes partes:

- Un analizador encargado de leer los ficheros TestSpec.
- Un generador encargado de obtener los valores aleatorios una vez obtenida la especificación.
- Un formateador encargado de volcar el conjunto de casos de prueba generado al formato que se desee.

TestGenerator-Autoseed utiliza elementos de las 3 partes por las que está formada TestGenerator, pero es una herramienta separada.

La herramienta **TestGenerator-Autoseed** posee las siguientes dependencias:

- Módulos `test-generator`, `test-generator-xtext`, `bpel-packager`, `bpel-xmlbeans` y `xpath-parser` del repositorio del proyecto *WS-BPEL Testing Tools* [50] del grupo UCASE.
- Dependencias externas como Java 6 [39], BPELUnit 1.6 [31], JUnit 4 [27] o JOptSimple [44].

4.1.2. Interfaz de usuario

TestGenerator-Autoseed no posee GUI. La herramienta se emplea a través de la línea de órdenes:

```
test-generator-autoseed (argumentos).
```

En el apéndice B están disponibles los comandos de los que dispone **TestGenerator-Autoseed**, además de los argumentos necesarios.

4.2. Funciones

Las funciones de **TestGenerator-Autoseed** son:

- Generar un conjunto de casos de prueba aplicando la siembra automática dada una composición WS-BPEL y una especificación en formato TestSpec.
- Generar un conjunto de casos de prueba aplicando la siembra automática optimizada dada una composición WS-BPEL, una especificación en formato TestSpec y una especificación del formato de los casos de prueba para BPELUnit.
- Contar el número de casos de prueba que se generarían aplicando la técnica de siembra automática a una composición WS-BPEL y una especificación TestSpec.
- Contar el número de casos de prueba que se generarían aplicando la técnica de siembra automática optimizada a una composición WS-BPEL, una especificación TestSpec y una especificación del formato de los casos de prueba para BPELUnit.

4.3. Características del usuario

Dado el objetivo de esta herramienta, que es el de generar un conjunto de casos de prueba aplicando la siembra automática para una composición WS-BPEL y utilizarlo a la hora de realizar estudios de prueba de mutaciones con MuBPEL, es necesario que el usuario conozca como mínimo:

- El lenguaje WS-BPEL 2.0 [37].
- El lenguaje de especificación TestSpec [47].
- El framework de pruebas BPELUnit [31].
- La técnica de mutaciones [25].

4.4. Restricciones generales

4.4.1. Control de versiones

Para el desarrollo de este PFC, se ha empleado un sistema de control de versiones, lo cual tiene las siguientes ventajas:

- Sirve para respaldar los datos que se poseen del proyecto.
- Permite revertir cambios perjudiciales en el proyecto, ya que se disponen de todas las versiones del mismo.

Subversion

En concreto, se ha empleado el sistema de control de versiones Subversion.

Este SCV es un sistema centralizado, ya que los cambios son enviados a un servidor, ampliamente utilizado por desarrolladores.

Las órdenes más útiles a la hora de emplear este SCV son las siguientes:

- Para descargar una copia de trabajo de un repositorio existente, se ejecuta:
  `svn co`

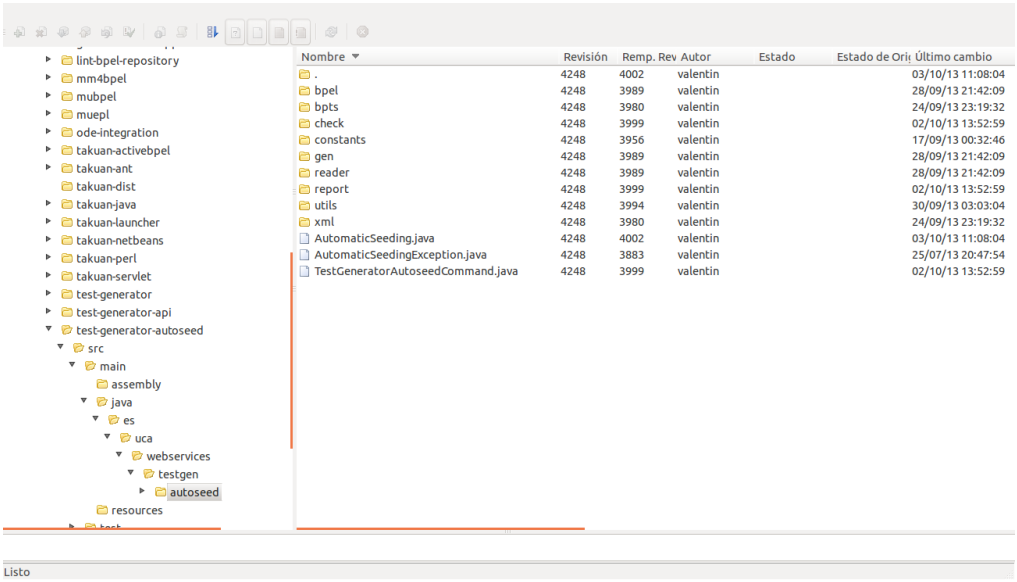


Figura 4.1.: RapidSVN

- Para actualizar la copia de trabajo, se ejecuta:
  `svn up`

- Para enviar los cambios realizados localmente, se ejecuta:
  `svn ci –m "Cambios realizados"`
  Mediante la opción -m se indica un mensaje con los cambios que se han realizado en la revisión enviada.

Para obtener más información sobre éstas y otras órdenes disponibles, se puede consultar el manual [11].

Éste es el manejo de Subversion mediante línea de órdenes. Si se desea un manejo más cómodo, existen varias interfaces gráficas que permiten una mayor comodidad a la hora de interactuar con el mismo. En este caso, se empleó RapidSVN. Podemos ver en la figura 4.1 una instantánea de la misma.

Esta interfaz hace más sencillo el manejo de Subversion, y permite interactuar con el repositorio de manera más intuitiva, dando la opción de seleccionar los cambios que queramos actualizar de manera rápida y simple. Para obtener la última versión, se puede visitar su sitio web [33].

4.4.2. Servidor de integración continua

La integración continua es una metodología propuesta por Martin Fowler [21] que consiste en integrar los cambios realizados en un proyecto software con el objeto de detectar los fallos lo más pronto posible.

Para llevar a cabo esta metodología, se implanta un servidor de integración continua el cual, cada cierto tiempo, realiza lo siguiente:

1. Obtener, mediante el SCV correspondiente, la última versión del código fuente.
2. Compilar el código obtenido y ejecutar las pruebas unitarias.
3. Generar un informe con los resultados.

Para realizar la compilación y ejecutar las pruebas, el servidor se apoya en herramientas como Apache Maven, las cuales se encargan de la gestión de los proyectos software.

Jenkins

En concreto, se ha empleado el servidor de integración continua Jenkins [10]. En la figura 4.2 podemos ver cómo se organiza dicho servidor.

Este servidor se encarga de realizar las tareas explicadas anteriormente, cada varias horas y cada vez que se registran nuevos cambios en el código. Una vez finaliza su trabajo, en el caso de existir algún error, ya sea de compilación o porque exista alguna prueba que no haya sido superada, envía un correo al desarrollador para avisarle de que esta versión no es estable, indicando el porqué para que éste pueda arreglar los fallos existentes.

4.4.3. Gestor de repositorio

A la hora de realizar un proceso de desarrollo en grupo es importante gestionar de forma adecuada el control de las bibliotecas que se utilizan dentro del proyecto para evitar incompatibilidades de licencias, uso inadecuado de bibliotecas o el uso de diferentes versiones de la misma biblioteca a la vez. Esta gestión suele llevarse a cabo a través de herramientas concretas, denominadas gestores de repositorio.

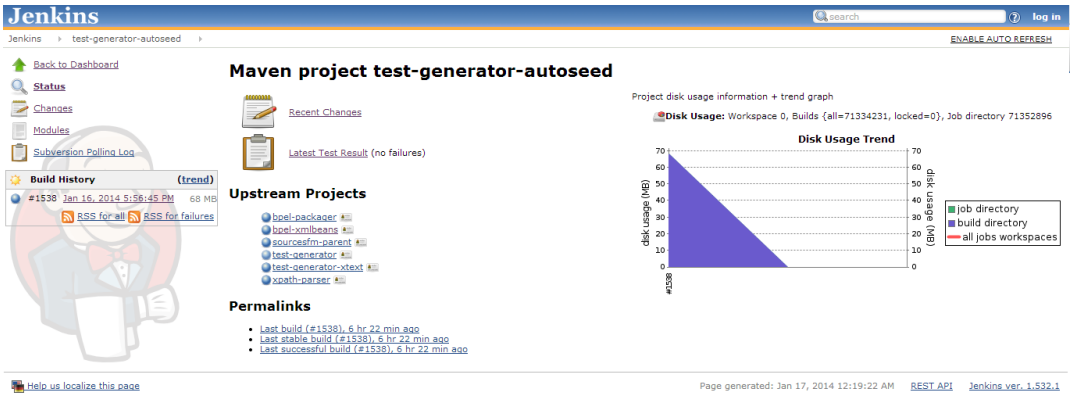


Figura 4.2.: Jenkins

Nexus

En este PFC se ha utilizado el gestor de repositorios Nexus [43]. Este gestor se integra perfectamente con Apache Maven [17], herramienta utilizada para la gestión del ciclo de vida del software, lo que lo hace idóneo para su uso.

Otras bondades de Nexus son la facilidad de configuración y su modelo de seguridad, el más robusto entre los gestores actuales.

4.4.4. Calidad del código fuente

El uso de métricas para el control de la calidad del código fuente es bastante útil a la hora de obtener un código más robusto. Mediante estas métricas, se pueden obtener estadísticas acerca del número de clases, métodos y atributos, permitiendo un empleo óptimo de las mismas.

Sonar

Una herramienta que sirve para la generación de métricas para código fuente Java es Sonar. En la figura 4.3 vemos la estructura principal de esta aplicación.

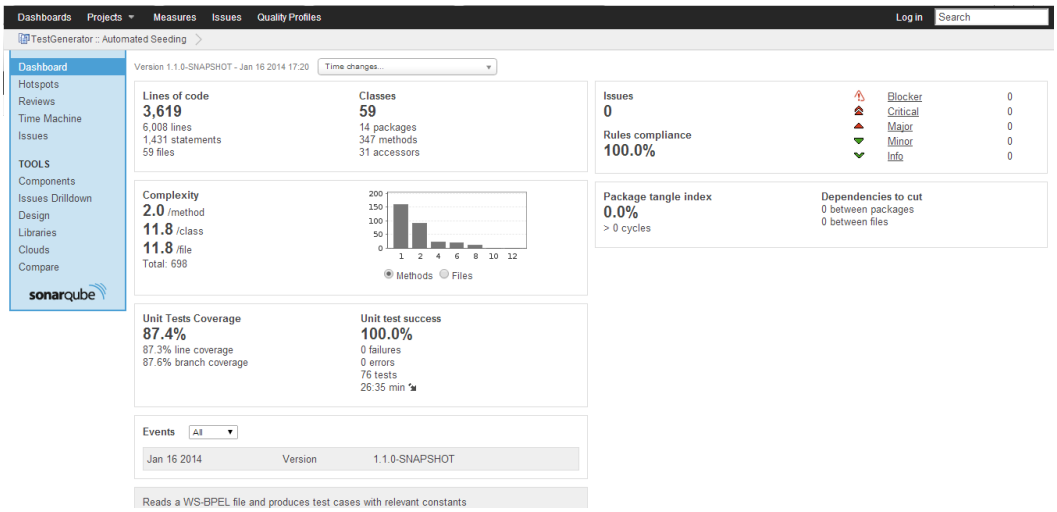


Figura 4.3.: Sonar

Esta herramienta muestra los errores que existen dentro del código. Los errores están clasificados como *error* (*blocker*, *critical*), *warning* (*major*, *minor*) e *info*. Están ordenados en función de su importancia, siendo el error *blocker* el más grave, y el error *info* el menos importante.

También se muestran estadísticas acerca de las líneas de código escritas, del número de clases empleadas y de las dependencias que tienen las clases, entre otros.

4.4.5. Lenguajes de programación y tecnologías

Los lenguajes de programación y las tecnologías que se han utilizado durante el desarrollo de este PFC son:

- WSDL: Se emplea para la descripción de servicios Web.
- SOAP: Se emplea para el intercambio de mensajes de forma distribuida.
- WS-BPEL: Se emplea para la composición de servicios Web.
- XPath: Se emplea para el recorrido de documentos XML.
- XML Schema: Se emplea para la descripción de la estructura de documentos XML.

- Java: Se emplea para el desarrollo de las clases que implementan la herramienta.
- JUnit: Se emplea para la realización de pruebas unitarias en clases Java.
- BPELUnit: Se emplea para la realización de pruebas unitarias en composiciones WS-BPEL.

4.4.6. Herramientas

Para desarrollar este PFC se han utilizado las siguientes herramientas:

- Apache Maven [17]: Sistema de gestión de proyetos software, encargado de compilar, ejecutar pruebas y obtener la documentación correspondiente.
- Eclipse [20]: Entorno de desarrollo multiplataforma empleado principalmente en el desarrollo de aplicaciones Java, aunque existen multitud de extensiones que le permiten ser útil para el desarrollo bajo cualquier lenguaje de programación.
- Meld [51]: Visor de diferencias gráfico.
- XMLEye [13]: Visor de documentos XML gráfico, que permite navegar con gran comodidad a través de los mismos.
- Xacobeo [41]: Aplicación que permite ejecutar consultas XPath de manera gráfica, muy útil para comprobar la corrección y validez de las mismas.

4.4.7. Sistemas operativos y hardware

Este PFC ha sido desarrollado y probado en GNU-Linux, empleándose en concreto la distribución Ubuntu, en su versión 13.04.

En cuanto a los requisitos hardware, se recomienda ejecutar la herramienta TestGenerator-Autoseeden un ordenador que disponga de una memoria RAM de, al menos, 2 GiB de capacidad, ya que para composiciones WS-BPEL grandes, la herramienta consume bastantes recursos.

5. Desarrollo del proyecto

En este capítulo se recoge el proceso de desarrollo, es decir, las fases de análisis, diseño, implementación y prueba de este PFC. Cabe destacar que las fases de análisis y diseño recogen la última iteración del producto, únicamente, al ser la que contiene la arquitectura del programa al completo.

5.1. Modelo de ciclo de vida

El modelo de ciclo de vida elegido para este PFC es el modelo de ciclo de vida incremental. En la figura 5.1 podemos ver las principales características de este modelo. Está basado en el modelo lineal-secuencial, añadiéndole la capacidad de evolucionar, ya que las fases de análisis, diseño, implementación y prueba no sólo ocurren una vez.

Se parte de unos requisitos iniciales, y tras cada iteración, se obtiene un producto software que puede ser utilizado. No es necesario esperar al final del proceso para obtener un producto software, al contrario que en el modelo lineal-secuencial. Además, no es necesario que los requisitos estén completamente definidos al comienzo, pues pueden redefinirse tras cada iteración, permitiendo añadir nuevas funcionalidades al software.

Este modelo se adapta perfectamente a nuestras necesidades, ya que en cada iteración se añaden nuevas funcionalidades a la herramienta. Tras las fases de análisis, diseño, implementación y prueba, obtendremos una nueva versión de **TestGenerator-Autoseed**, a la cual se le ha añadido nueva funcionalidad. De esta forma, podemos utilizar la herramienta sin tener que esperar a que estén definidos todos los requisitos.

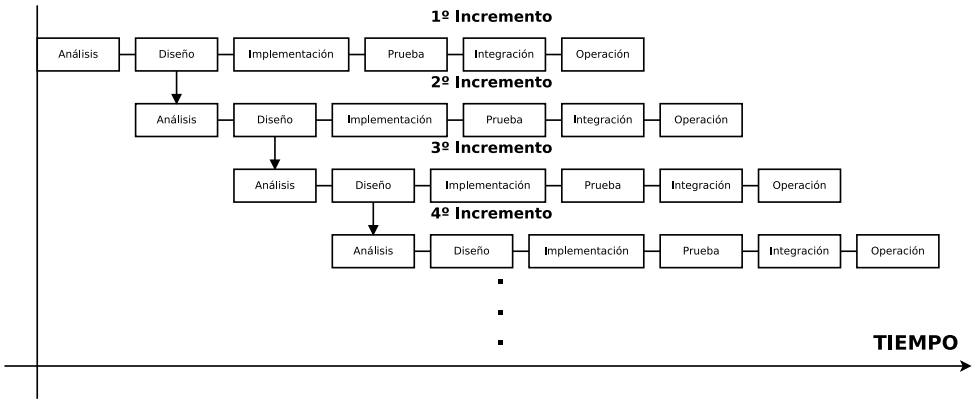


Figura 5.1.: Modelo de ciclo de vida incremental

5.2. Herramienta de modelado empleada: *dia*

La herramienta *dia* nos permite dibujar los diferentes tipos de diagramas UML existentes, como los diagramas de casos de uso, de clases o de secuencia. Además, permite dibujar otro tipo de diagramas, como los de entidad-relación o de flujo.

Tiene la ventaja de ser personalizable, pues mediante un archivo XML se pueden añadir nuevas formas que podemos emplear en nuestros diagramas, de una manera análoga a un archivo *svg*.

Los diagramas generados con esta herramienta pueden ser exportados a una gran cantidad de formatos, como por ejemplo *png*, *svg*, *eps*, o directamente a código LaTeX.

Además, se puede generar el código Java, C++, Pascal y Python correspondiente a los diagramas generados, exportándolos directamente.

La herramienta *dia* está disponible para Windows, Linux y Mac OS X, y se puede descargar en: `http://dia-installer.de/`.

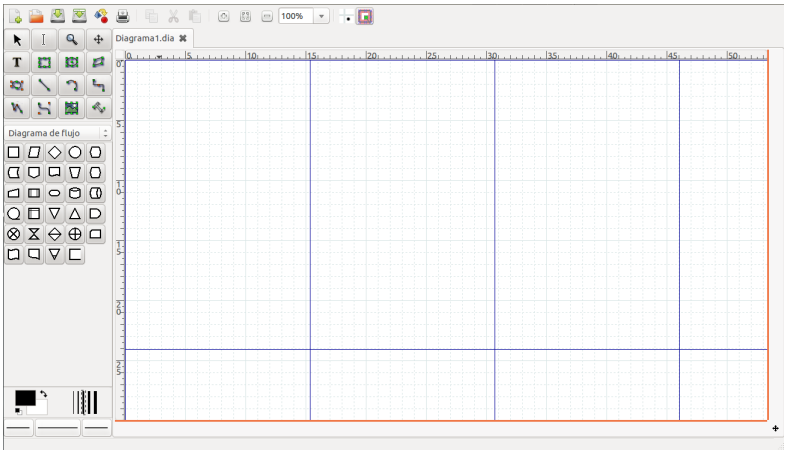


Figura 5.2.: Herramienta de modelo *dia*

5.3. Requisitos

5.3.1. Funcionales

- Generar un conjunto de casos de prueba mediante siembra automática dados una composición WS-BPEL y una especificación TestSpec. Como resultado de esta operación se debe generar un fichero en formato VTL, el cual contiene los valores de las variables por cada caso de prueba. De manera opcional, se puede generar un fichero en el que se detalle información relevante sobre el proceso de siembra automática realizado.
- Obtener el número de casos de prueba necesarios para aplicar la siembra automática dados una composición WS-BPEL y una especificación TestSpec. Como resultado de esta operación se debe mostrar el número de casos de prueba calculado. De manera opcional, se puede generar un fichero en el que se detalle información relevante sobre el proceso de cálculo realizado.
- Generar un conjunto de casos de prueba mediante siembra automática optimizada dados una composición WS-BPEL, una especificación TestSpec y un conjunto de casos de prueba BPELUnit. Como resultado de esta operación se debe generar un fichero en formato VTL, el cual contiene los valores de las variables por cada caso de prueba. De manera opcional, se puede generar un fichero en el

que se detalle información relevante sobre el proceso de siembra automática realizado.

- Obtener el número de casos de prueba necesarios para aplicar la siembra automática optimizada dados una composición WS-BPEL, una especificación TestSpec y un conjunto de casos de prueba BPELUnit. Como resultado de esta operación se debe mostrar el número de casos de prueba calculado. De manera opcional, se puede generar un fichero en el que se detalle información relevante sobre el proceso de cálculo realizado.

5.3.2. De información

- El sistema debe guardar información relativa al proceso de siembra automática, concretamente:
  - El conjunto de constantes halladas en la composición WS-BPEL. Para cada constante, necesitamos saber su tipo, su valor y la variable de la composición con la que se relaciona.
  - El conjunto de variables de tipo básico de la especificación TestSpec, localizando las que estén dentro de estructuras complejas de listas y tuplas. Para cada una de ellas, se necesita saber su nombre y tipo.
  - El conjunto de variables de la especificación TestSpec compatibles con alguna de las constantes halladas.
- En caso de aplicar la siembra automática optimizada, el sistema, además de la información anterior, debe almacenar de forma adicional el conjunto de variables de la composición WS-BPEL relacionadas con cada variable de la especificación TestSpec. De cada variable de la composición se necesita saber únicamente la expresión de la misma.
- La composición WS-BPEL, el fichero `.spec` con la especificación TestSpec, el fichero `.vm` en formato VTL con el conjunto de valores para los casos de prueba y, en el caso de la siembra automática optimizada, el fichero `.bpts` con el conjunto de casos de prueba BPELUnit, se deben almacenar en ficheros, guardando su nombre y ruta.

5.3.3. De reglas de negocio

Cada caso de prueba generado mediante la herramienta contendrá únicamente una modificación, dadas una variable y una constante, respecto del caso de prueba aleatorio base.

5.3.4. De interfaz

La herramienta **TestGenerator-Autoseed** se utiliza dentro del estudio de prueba de mutaciones, en concreto como valores para las plantillas de un conjunto de casos de prueba BPELUnit de una composición WS-BPEL, participando en la ejecución de la misma con la herramienta **MuBPEL**.

Por lo tanto, los resultados generados por **TestGenerator-Autoseed** deben de respetar el formato VTL requerido por BPELUnit para el conjunto de valores de las variables de plantillas Apache Velocity.

5.3.5. No funcionales

- Rendimiento. Se busca un rendimiento alto, especialmente a la hora de generar conjuntos de casos de prueba para composiciones WS-BPEL con gran cantidad de constantes y especificaciones TestSpec con gran cantidad de variables o con variables cuya estructura sea muy compleja.

- Fiabilidad. Los conjuntos de valores generados han de ser correctos y respetar tanto las especificaciones TestSpec como la filosofía de la técnica de siembra automática.

- Mantenibilidad. Se busca que la herramienta sea fácil de mantener y de ampliar, con vistas de añadir funcionalidad futura de manera cómoda.

- Transportabilidad. La herramienta debe funcionar correctamente independientemente del SO donde se ejecute.

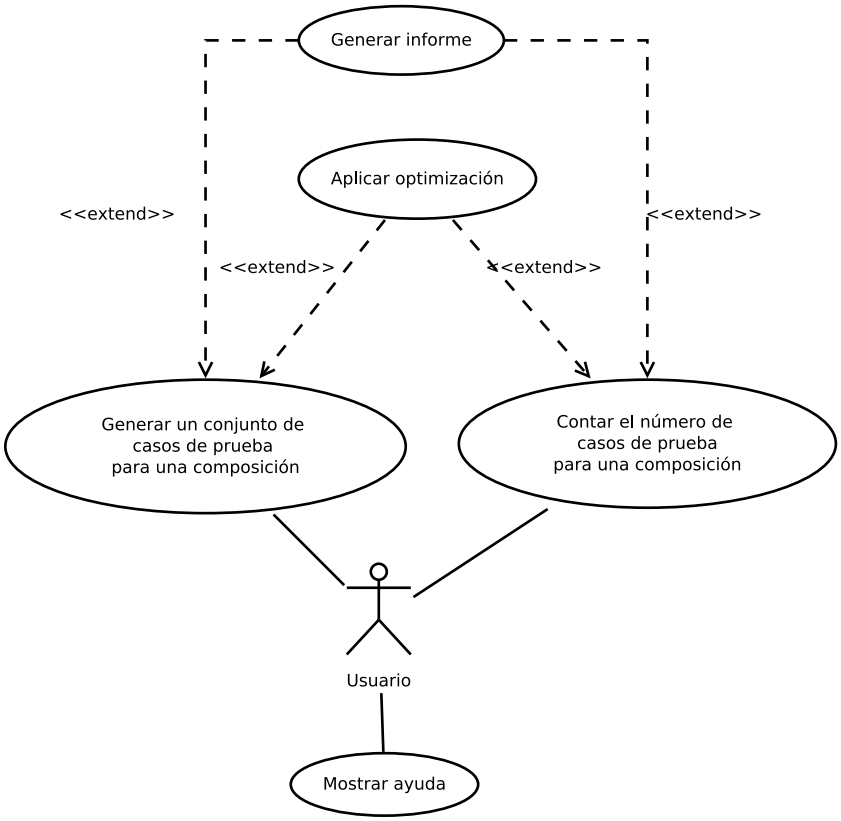


Figura 5.3.: Diagrama de casos de uso

5.4. Análisis

5.4.1. Modelo de casos de uso

Mediante el modelo de casos de uso, podemos reflejar cómo el usuario interactúa con el sistema, además de las funcionalidades del mismo.

En la figura 5.3 podemos ver el diagrama de casos de uso de **TestGenerator-Autoseed**.

A continuación, se facilitan las especificaciones de cada uno de los casos de uso del sistema.

5.4.1.1. Generar un conjunto de casos de prueba para una composición

Actor principal El usuario, quien desea generar un conjunto de casos de prueba para la composición aplicando la siembra automática.

Precondiciones Es necesario que exista una composición WS-BPEL y una especificación TestSpec asociada a la misma.

Postcondiciones Se obtiene un fichero en formato VTL con los valores de cada variable de la especificación para cada caso de prueba generado.

Escenario principal

1. El usuario selecciona la opción de generar el conjunto de casos de prueba, indicando la composición y la especificación.
2. El sistema genera un conjunto de valores para las variables de la especificación por cada caso de prueba siguiendo la técnica de siembra automática.

Extensiones

1a. El usuario selecciona la opción de generar el conjunto de casos de prueba con optimización, indicando la composición, la especificación y un conjunto de casos de prueba BPELUnit.

   1. El sistema realiza el caso de uso Aplicar optimización (ver 5.4.1.4).
   2. El caso de uso continúa.

1b. El usuario selecciona adicionalmente la opción de generar el informe relativo al proceso de siembra automática.

   1. El sistema realiza el caso de uso Generar informe (ver 5.4.1.5).
   2. El caso de uso continúa.

5.4.1.2. Contar el número de casos de prueba para una composición

Actor principal El usuario, quien desea contar el número de casos de prueba que se generarían aplicando la siembra automática.

Precondiciones Es necesario que exista una composición WS-BPEL y una especificación TestSpec asociada a la misma.

Postcondiciones Se obtiene el número de casos de prueba que se generarían aplicando la siembra automática dada la composición y la especificación.

Escenario principal

1. El usuario selecciona la opción de contar el número de casos de prueba, indicando la composición y la especificación.
2. El sistema muestra el número de casos de prueba que se generaría aplicando la técnica.

Extensiones

1a. El usuario selecciona la opción de contar el número de casos de prueba con optimización, indicando la composición, la especificación y un conjunto de casos de prueba BPELUnit.

   1. El sistema realiza el caso de uso Aplicar optimización (ver 5.4.1.4).
   2. El caso de uso continúa.

1b. El usuario selecciona adicionalmente la opción de generar el informe relativo al proceso de siembra automática.

   1. El sistema realiza el caso de uso Generar informe (ver 5.4.1.5).
   2. El caso de uso continúa.

5.4.1.3. Mostrar ayuda

Actor principal El usuario, quien desea saber las funciones del sistema y los parámetros que necesita para cada una de ellas.

Precondiciones Ninguna.

Postcondiciones El sistema muestra la ayuda, detallando las opciones y parámetros necesarios.

Escenario principal

1. El usuario selecciona la opción de ayuda.
2. El sistema muestra la ayuda, detallando las opciones y parámetros necesarios.

5.4.1.4. Aplicar optimización

Actor principal Debido a que es un caso de uso abstracto, no es iniciado por ningún actor. Se activa opcionalmente al realizar los casos de uso de generar un conjunto de casos de prueba (ver 5.4.1.1) o contar el número de casos de prueba (ver 5.4.1.2).

Precondiciones Es necesario que exista una composición WS-BPEL, una especificación TestSpec y un conjunto de casos de prueba BPELUnit asociados a la misma.

Postcondiciones Se obtienen las relaciones existentes entre las variables de la composición y las de la especificación.

Escenario principal

1. El sistema genera las relaciones existentes entre las variables de la composición y las de la especificación.

Extensiones

1a. No se puede aplicar la optimización.

1. El sistema muestra el error y continúa.

5.4.1.5. Generar informe

Actor principal Debido a que es un caso de uso abstracto, no es iniciado por ningún actor. Se activa opcionalmente al realizar los casos de uso de generar un conjunto de casos de prueba (ver 5.4.1.1) o contar el número de casos de prueba (ver 5.4.1.2).

Precondiciones Es necesario que exista una composición WS-BPEL y una especificación TestSpec asociada a la misma.

Postcondiciones Se obtiene un informe con información relativa al proceso de siembra automática realizado.

Escenario principal

1. El usuario selecciona el nivel de detalle *básico*.
2. El usuario selecciona el formato de salida en texto plano.
3. El sistema genera el informe solicitado.

Extensiones

1a. El usuario selecciona el nivel de detalle *intermedio*.

1b. El usuario selecciona el nivel de detalle *completo*.

2a. El usuario selecciona el formato de salida en XML.

2b. El usuario selecciona el formato de salida en YAML.

5.4.2. Modelo conceptual de datos

Mediante el modelo conceptual de datos, obtenemos cuáles son las clases que componen nuestro sistema, así como las relaciones entre las mismas.

En la figura 5.4 podemos ver el diagrama de clases conceptuales correspondiente a la herramienta **TestGenerator-Autoseed**.

Una *Composición* posee un nombre y una ubicación en disco. Está compuesta por un conjunto de instancias de *Constante* y por un conjunto de instancias de *VariableComposicion*. Una *Constante* tiene un nombre y un tipo de datos y una *VariableComposicion* tiene un nombre y un contenido, que es la estructura de la misma.

Una *Especificación* posee un nombre y una ubicación en disco, al igual que una *Composición*. Está compuesta por un conjunto de *VariableEspecificación*, las cuales poseen un nombre y un tipo.

Un *ConjuntoCasosPrueba* posee un nombre y una ubicación en disco, al igual que la *Composición* y la *Especificación* y se compone de un conjunto de *Plantilla*, las cuales tienen un contenido.

En caso de que se active la optimización, cada *VariableEspecificación* se relacionará con una o más instancias de *VariableComposición* a través de una *Plantilla*.

Para recopilar las instancias de *VariableEspecificación* compatibles con instancias de *Constante*, se crean instancias de *EntradaInforme*, las cuales componen a un *Informe* completo que contiene el nivel de detalle del mismo y el número de casos de prueba a generar, calculado a partir de las instancias de *EntradaInforme*.

Tomando el *Informe*, el *Generador*, que posee un formato de salida, es capaz de obtener los *Resultados*, que se crean con un nombre y una ubicación en disco.

5.4.3. Diagramas de secuencia

A continuación, se presentan los diagramas de secuencia correspondientes a cada caso de uso detallado anteriormente. Mediante estos diagramas, podemos identificar de manera intuitiva las operaciones que tendrá

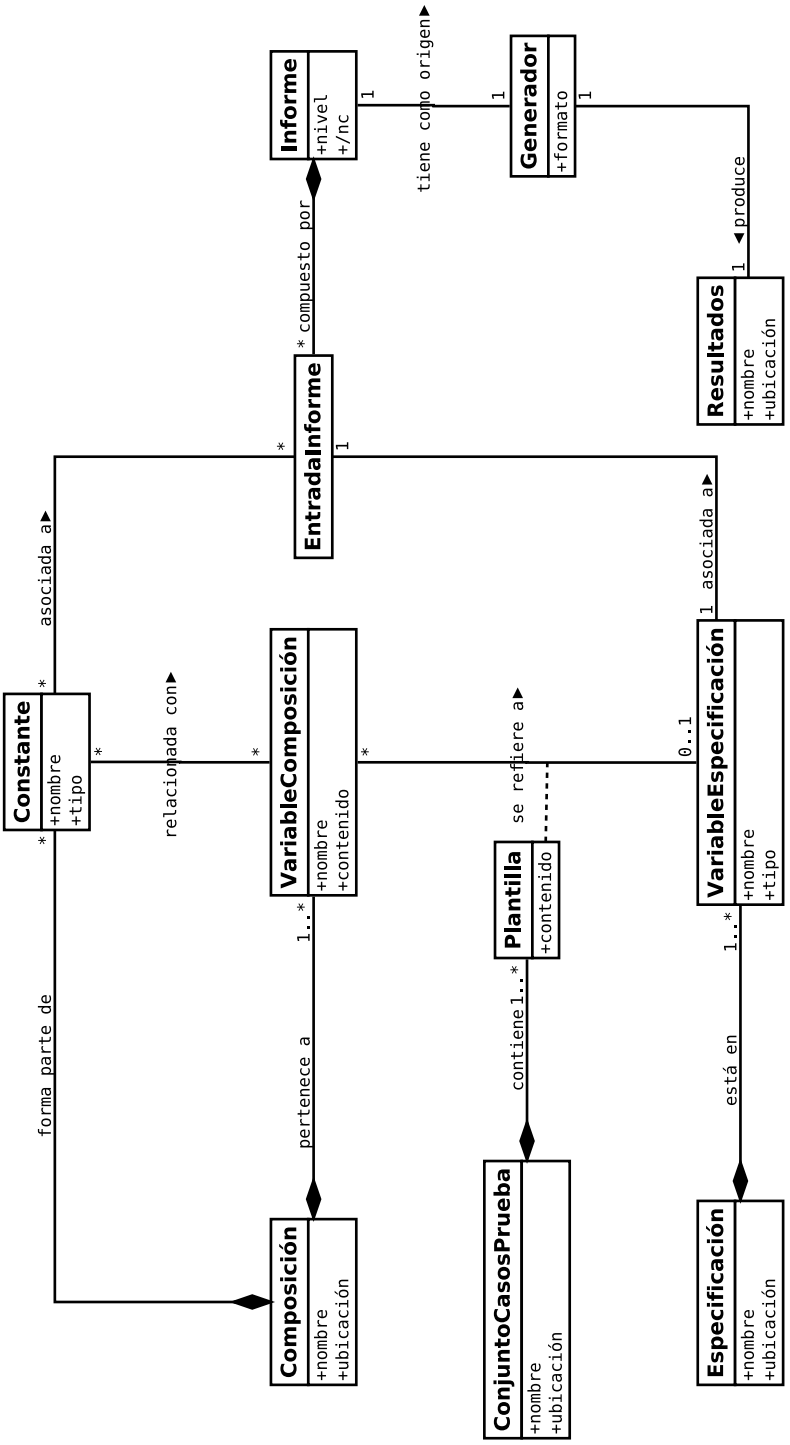


Figura 5.4.: Diagrama de clases conceptuales

nuestro sistema. Así pues, una vez identificadas, se añaden los contratos de las operaciones más relevantes.

5.4.3.1. Generar un conjunto de casos de prueba para una composición

En la figura 5.5 podemos ver el diagrama de secuencia correspondiente a este caso de uso.

Operación refComp := SeleccionarComposición(nombre)

Responsabilidades Selecciona una composición WS-BPEL existente.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una composición WS-BPEL cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refComp* a un objeto Composición, que representa a la composición seleccionada.

Operación refSpec := SeleccionarEspecificación(nombre)

Responsabilidades Selecciona una especificación TestSpec existente.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una especificación TestSpec cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refSpec* a un objeto Especificación, que representa a la especificación seleccionada.

Operación GenerarCasos(refComp, refSpec)

Responsabilidades Genera un conjunto de casos de prueba mediante siembra automática a partir de una composición WS-BPEL y una especificación TestSpec.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existen una referencia a objeto Composición *refComp* y una referencia a objeto Especificación *refSpec*.

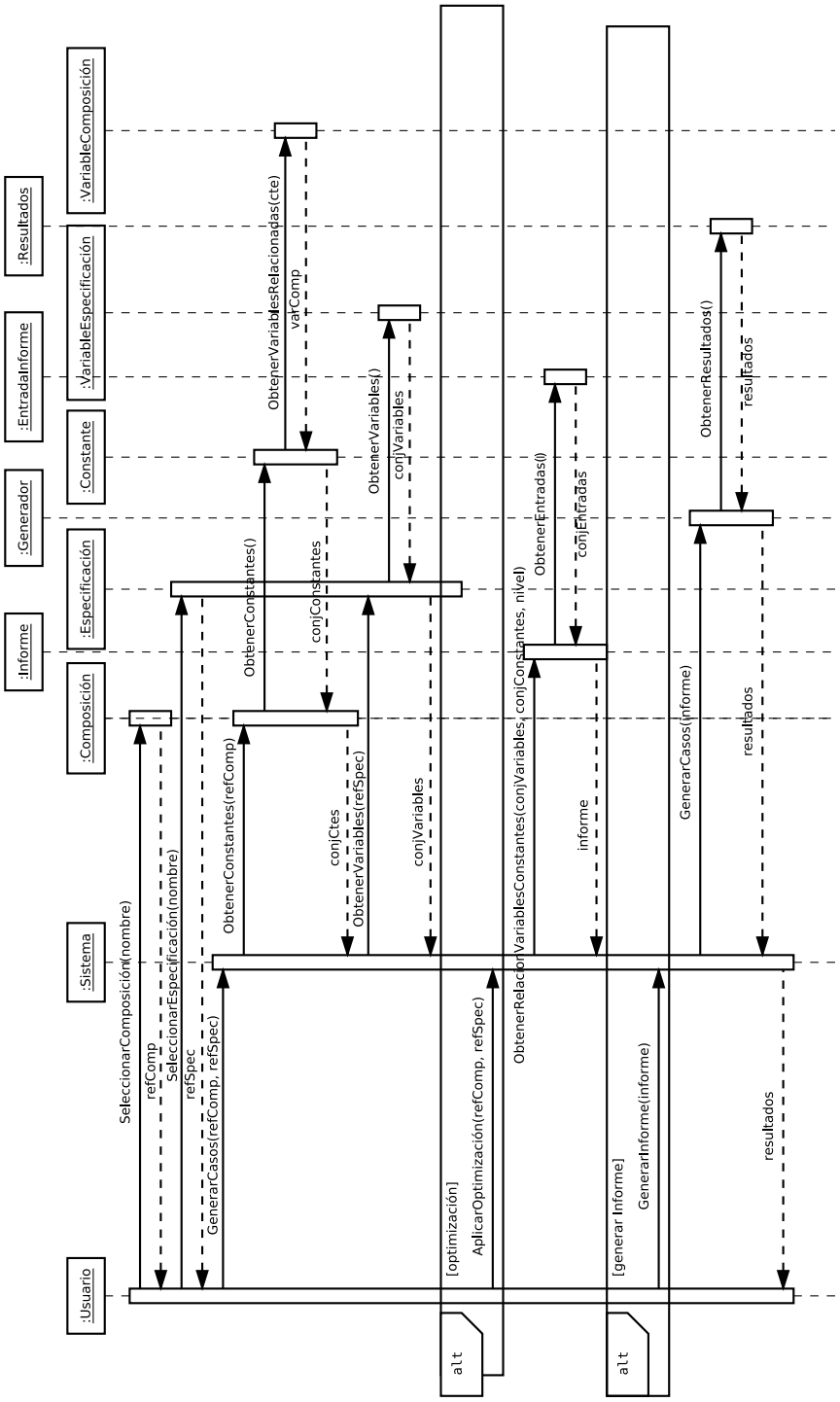


Figura 5.5.: Diagrama de secuencia de Generar un conjunto de casos de prueba para una composición

Postcondiciones Se obtiene el conjunto de casos de prueba de la composición y especificación dadas.

Operación conjConstantes := ObtenerConstantes(refComp)

Responsabilidades Obtiene el conjunto de constantes existentes en la composición dada.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia a objeto Composición *refComp*.

Postcondiciones Se obtiene el conjunto de constantes de la composición dada.

Operación varComp := ObtenerVariablesRelacionadas(cte)

Responsabilidades Obtiene el conjunto de variables de la composición relacionadas con la constante dada.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia a objeto Constante *cte*.

Postcondiciones Se obtiene el conjunto de variables de la composición relacionadas con la constante dada.

Operación conjVariables := ObtenerVariables(refSpec)

Responsabilidades Obtiene el conjunto de variables existentes en la especificación dada.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia a objeto Especificación *refSpec*.

Postcondiciones Se obtiene el conjunto de variables de la especificación dada.

Operación ObtenerRelacionVariablesConstantes(conjVariables, conjConstantes, nivel)

Responsabilidades Obtiene, por cada variable de la especificación, el conjunto de constantes compatibles con ella.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existen un conjunto de variables de la especificación *conjVariables* y un conjunto de constantes *conjConstantes*.

Postcondiciones Se obtiene un diccionario en el que, por cada variable de pa especificación, el conjunto de constantes compatibles con ella.

Operación GenerarCasos(informe)

Responsabilidades Obtiene el conjunto de casos de prueba dado el conjunto de relaciones halladas.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe un diccionario de relaciones *informe*.

Postcondiciones Se obtiene un conjunto de casos de prueba acorde al diccionario recibido.

5.4.3.2. Contar el número de casos de prueba para una composición

En la figura 5.6 podemos ver el diagrama de secuencia correspondiente a este caso de uso.

Operación ContarCasos(refComp, refSpec)

Responsabilidades Cuenta el número de casos de prueba que se generarían mediante siembra automática a partir de una composición WS-BPEL y una especificación TestSpec.

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Existen una referencia a objeto Composición *refComp* y una referencia a objeto Especificación *refSpec*.

Postcondiciones Se obtiene el número de casos de prueba para la composición y especificación dadas.

Operación ContarCasos(informe)

Responsabilidades Obtiene el número de casos de prueba dado el conjunto de relaciones halladas.

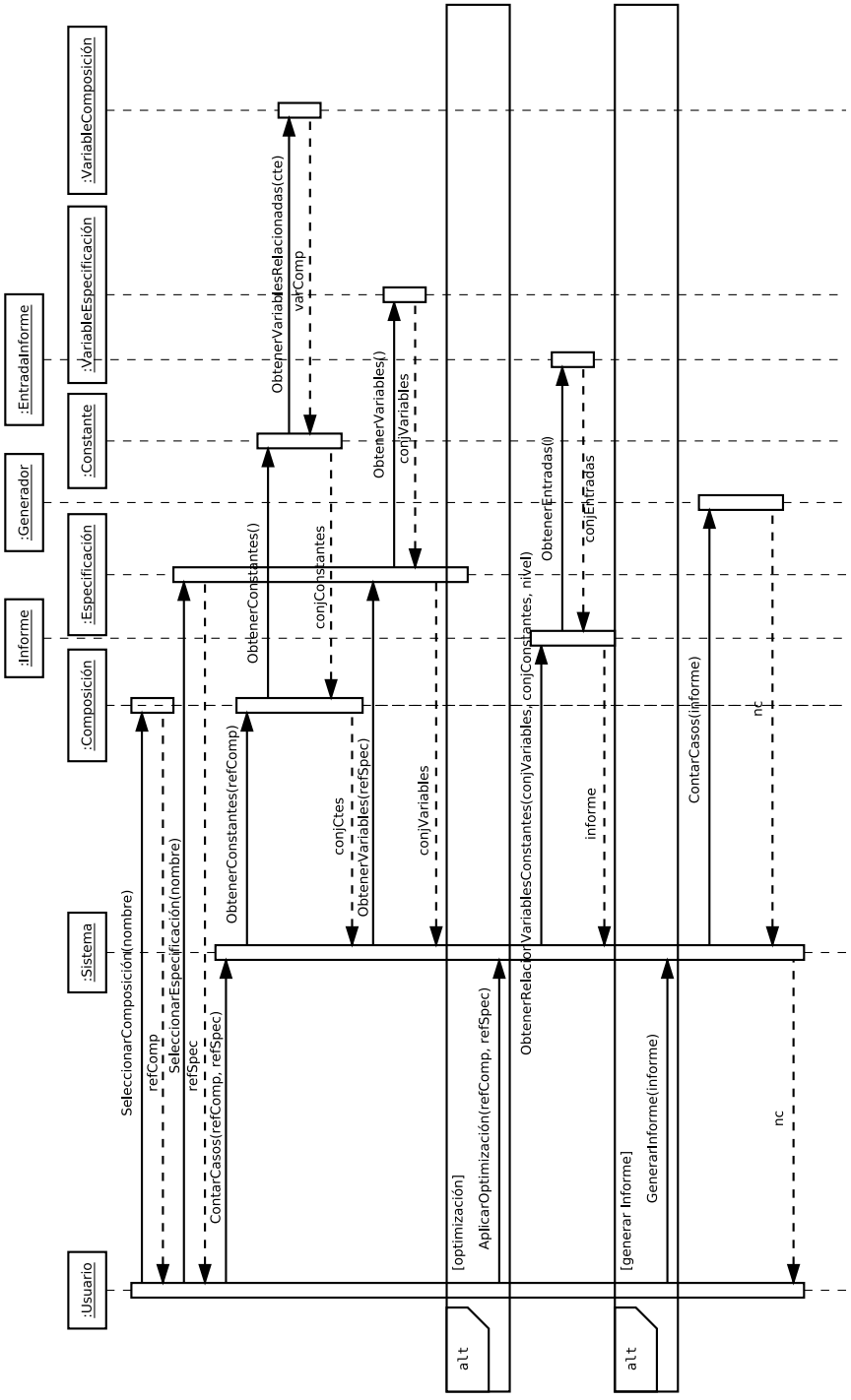


Figura 5.6.: Diagrama de secuencia de Contar el número de casos de prueba para una composición

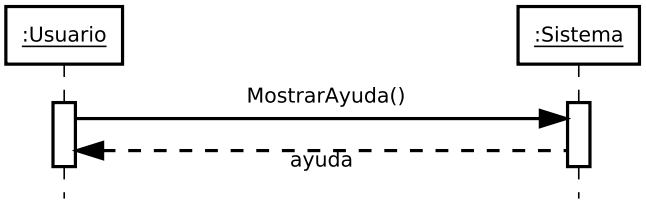


Figura 5.7.: Diagrama de secuencia de Mostrar ayuda

Referencias cruzadas Véase 5.4.1.2.

Precondiciones Existe un diccionario de relaciones *informe*.

Postcondiciones Se obtiene el número de casos de prueba acorde al diccionario recibido.

El resto de contratos de operaciones correspondientes a este caso de uso se encuentra disponible en la sección 5.4.3.1.

5.4.3.3. Mostrar ayuda

En la figura 5.7 podemos ver el diagrama de secuencia correspondiente a este caso de uso.

Operación MostrarAyuda()

Responsabilidades Muestra la ayuda del sistema.

Referencias cruzadas Véase 5.4.1.3.

Precondiciones Ninguna.

Postcondiciones El sistema muestra la ayuda, opciones disponibles y parámetros de las mismas.

5.4.3.4. Aplicar optimización

En la figura 5.8 podemos ver el diagrama de secuencia correspondiente a este caso de uso.

Operación refConj := SeleccionarConjuntoCasosPrueba(nombre)

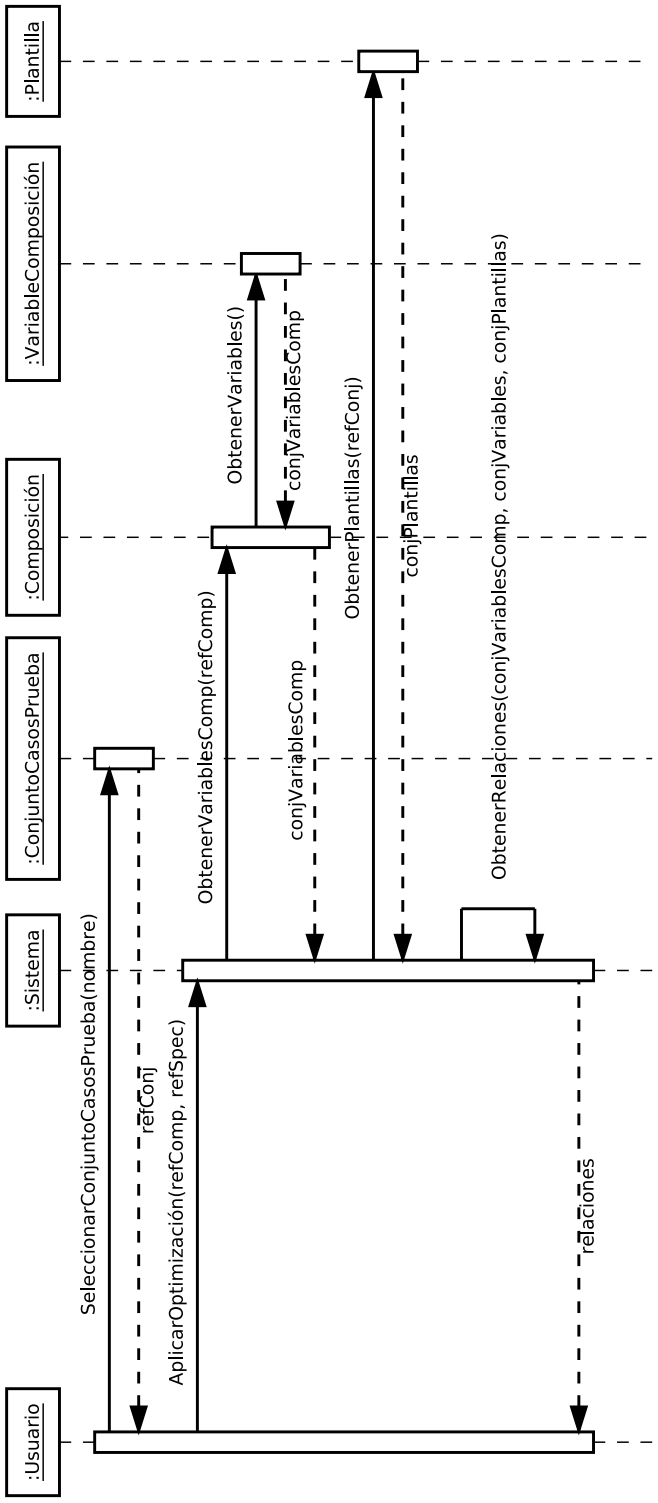


Figura 5.8.: Diagrama de secuencia de Aplicar optimización

Responsabilidades Selecciona un conjunto de casos de prueba BPELUnit existente.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existe un conjunto de casos de prueba BPELUnit cuyo nombre es igual a *nombre*.

Postcondiciones Se obtiene una referencia *refConj* a un objeto ConjuntoCasosPrueba, que representa al conjunto de casos de prueba seleccionado.

Operación AplicarOptimización(refComp, refSpec)

Responsabilidades Aplica la optimización de la técnica de siembra automática a partir de una composición WS-BPEL y una especificación TestSpec.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existen una referencia a objeto Composición *refComp* y una referencia a objeto Especificación *refSpec*.

Postcondiciones El sistema aplica la optimización a la hora de realizar el proceso de generación.

Operación conjVariables := ObtenerVariables(refComp)

Responsabilidades Obtiene el conjunto de variables existentes en la composición dada.

Referencias cruzadas Véase 5.4.1.1.

Precondiciones Existe una referencia a objeto Composición *refComp*.

Postcondiciones Se obtiene el conjunto de variables de la composición dada.

Operación conjPlantillas := ObtenerPlantillas(refConj)

Responsabilidades Obtiene el conjunto de plantillas existentes en en conjunto de casos de prueba dado.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existe una referencia a objeto ConjuntoCasosPrueba *refConj*.

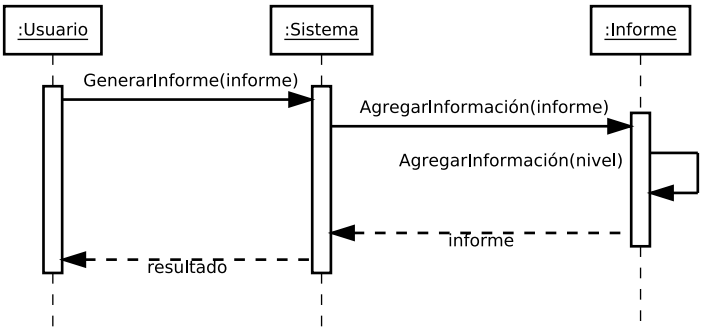


Figura 5.9.: Diagrama de secuencia de Generar informe

Postcondiciones Se obtiene el conjunto de plantillas del conjunto de casos de prueba dado.

Operación ObtenerRelaciones(conjVariablesComp, conjVariables, conjPlantillas)

Responsabilidades Obtiene las relaciones existentes entre las variables de la composición y las de la especificación.

Referencias cruzadas Véase 5.4.1.4.

Precondiciones Existen un conjunto de variables de la especificación *conjVariables*, un conjunto de variables de la composición *conjVariablesComp* y un conjunto de plantillas *conjPlantillas*.

Postcondiciones Se obtienen las relaciones entre las variables de la especificación y las de la composición y el sistema las añade como restricción a la hora de hallar las relaciones entre variables y constantes.

5.4.3.5. Generar informe

En la figura 5.9 podemos ver el diagrama de secuencia correspondiente a este caso de uso.

Operación GenerarInforme(informe)

Responsabilidades Genera el informe correspondiente al proceso de siembra automática.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existe una referencia a objeto Informe *informe*.

Postcondiciones El sistema muestra el informe asociado al proceso de siembra automática.

Operación AgregarInformación(informe)

Responsabilidades Agrega la información necesaria al informe.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existe una referencia a objeto Informe *informe*.

Postcondiciones Se agrega la información necesaria al informe del proceso.

Operación AgregarInformación(nivel)

Responsabilidades Agrega la información necesaria al informe en función del nivel de detalle.

Referencias cruzadas Véase 5.4.1.5.

Precondiciones Existe una referencia a objeto Informe *informe*.

Postcondiciones Se agrega la información necesaria al informe del proceso teniendo en cuenta el nivel de detalle establecido.

5.5. Diseño

En esta sección se decidirá la estructura interna que tendrá este PFC, así como las decisiones de diseño tomadas y el porqué de las mismas.

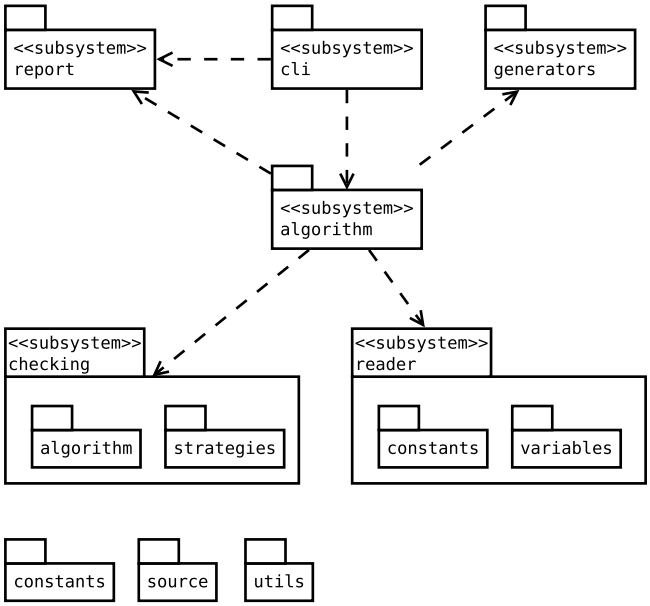


Figura 5.10.: Diagrama de paquetes de TestGenerator-Autoseed

5.5.1. Arquitectura del sistema

En la figura 5.10 podemos ver el conjunto de paquetes que conforma la herramienta TestGenerator-Autoseed. Por motivos de claridad, las relaciones de dependencia entre paquetes mostradas en el diagrama se refieren únicamente a subsistemas.

La herramienta TestGenerator-Autoseed se compone de una serie de bloques principales o subsistemas, que son los siguientes:

- Subsistema de interacción con el usuario (*cli*).
- Subsistema de control (*algorithm*).
- Subsistema de lectura (*reader*).
- Subsistema de chequeo (*checking*).
- Subsistema de generación de informes (*report*).
- Subsistema de generación de casos de prueba (*generators*).

Además de los bloques principales, la herramienta está compuesta por una serie de bloques secundarios, los cuales son:

- Bloque de gestión de fuentes (*source*).
- Bloque de gestión de constantes (*constants*).
- Bloque de utilidades (*utils*).

Los motivos de esta división se deben principalmente a lo siguiente:

- Facilidad a la hora de desarrollar la herramienta, puesto que el diseño en bloques permite agregar nueva funcionalidad de manera cómoda.
- Mayor eficacia en el momento de realizar las pruebas, puesto que la realización de pruebas por bloques permite la corrección de errores de una forma más eficaz.
- Posibilidad de reutilizar el código en futuros proyectos del grupo UCASE, algo que ya se está pensando en hacer, sin que sea necesario utilizar para ello el sistema entero.
- Permitir ampliar el sistema, de manera que añadir la posibilidad de trabajar con otros entornos o lenguajes no sea demasiado costosa en tiempo.

Dado que en el seno del grupo UCASE el lenguaje de programación elegido a la hora de desarrollar los proyectos es Java, el diseño, implementación y pruebas de esta herramienta irán enfocados hacia esa plataforma.

En la sección 5.6 se detallará la funcionalidad y se mostrará el diagrama de clases dependientes de la plataforma de cada uno de los bloques.

5.5.2. Restricciones de diseño

Cada uno de los bloques que componen la herramienta TestGenerator-Autoseed deben de ajustarse a una serie de restricciones, que son las siguientes:

- El bloque debe de tener un bajo acoplamiento respecto del resto de bloques principales, con el objetivo de una posible reutilización futura.

- En el caso de los bloques principales de los que depende el bloque principal *algorithm*, se debe de diseñar una interfaz para cada uno de ellos, con el objetivo de, en un futuro, poder aportar implementaciones de la técnica para otros lenguajes de programación y entornos con un esfuerzo pequeño.
- Las clases que correspondan a cada bloque deben tener una complejidad baja, con el objetivo de facilitar la depuración de errores y la mantenibilidad del sistema.

5.5.2.1. Patrones de diseño utilizados

Con el objetivo de garantizar el cumplimiento de las restricciones anteriores, se han utilizado una serie de patrones de diseño [23] existentes, los cuales iremos detallando a continuación.

Fachada El patrón Fachada (*Facade*) [23] es un patrón de tipo estructural que proporciona una interfaz común para un subsistema existente, con vistas a la facilidad de uso del mismo.

El uso de este patrón de diseño dentro de nuestros bloques nos permitirá por tanto ofrecer para cada uno de ellos una interfaz de uso, lo que tiene como ventaja el desacoplamiento de ésta respecto a la implementación que le aportemos, además del desacoplamiento entre los propios bloques.

En la figura 5.11 podemos ver el diagrama de clases que representa a la estructura del mismo.

En este caso, el cliente se comunica con un objeto de la clase *Fachada*, el cual se encarga de gestionar la petición del cliente a los objetos del subsistema que la fachada encapsula.

Estrategia El patrón Estrategia (*Strategy*) [23] es un patrón de comportamiento que permite intercambiar en tiempo de ejecución una serie de algoritmos en función a unos parámetros dados.

Dentro de nuestros bloques, el uso de este patrón permitirá simplificar el chequeo de restricciones, por ejemplo, delegando en este patrón el algoritmo de chequeo a usar en función del tipo de una constante y de

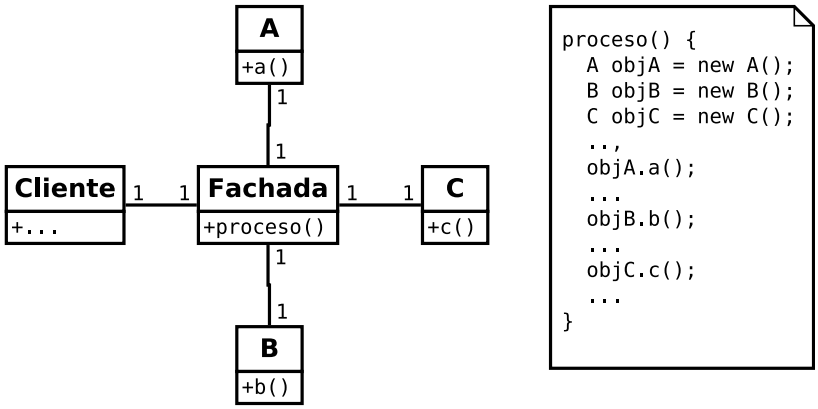


Figura 5.11.: Estructura del patrón Fachada

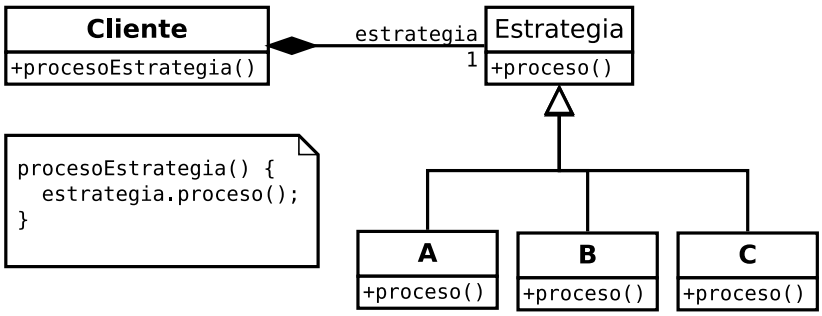


Figura 5.12.: Estructura del patrón Estrategia

una variable. También hace más simple el añadir nuevos algoritmos a los ya existentes, como podemos comprobar en la figura 5.12.

En este caso, el cliente crea o recibe la instancia apropiada de una de las subclases de la clase abstracta *Estrategia* y a la hora de llamar a su método `procesoEstrategia`, delega en la estrategia la implementación del mismo.

Método Fábrica El patrón Método Fábrica (*Factory Method*) [23] es un patrón estructural con el cual se consigue delegar la creación de los objetos de una jerarquía de clases a las subclases de la misma.

En la mayoría de bibliotecas y proyectos que utilizan el lenguaje Java aparece el uso de este patrón de diseño. En nuestro caso podremos apli-

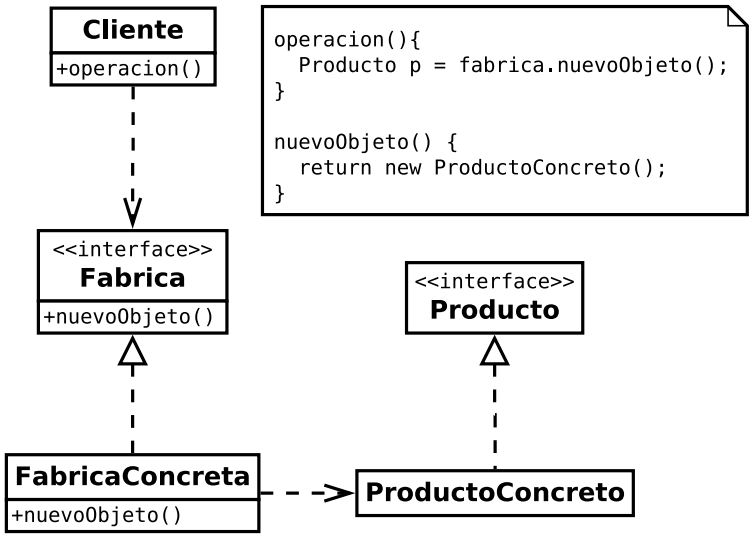


Figura 5.13.: Estructura del patrón Método Fábrica

carlo en muchos bloques de la herramienta, por ejemplo a la hora de crear las constantes en función de su tipo o a la hora de elegir el formato que queremos de salida en el bloque de generación de informes.

En la figura 5.13 podemos ver la estructura de este patrón de diseño.

Como podemos ver, el cliente, para la creación de los productos, acude a la fábrica. En función del tipo de fábrica que esté instanciada, se creará un tipo de producto concreto.

Con el objetivo de simplificar la aplicación de este patrón en muchas ocasiones el método `nuevoObjeto` se parametriza, utilizando el parámetro para crear los diferentes tipos de productos.

Instancia Única El patrón Instancia Única (*Singleton*) [23] es un patrón estructural empleado para garantizar la existencia de un único objeto de una clase en el sistema, y proveer el acceso al mismo.

En multitud de ocasiones, este patrón se suele aplicar de forma conjunta al patrón Método Fábrica (Ver 5.5.2.1), para asegurar la existencia de una única instancia del objeto que juega el papel de fábrica.

A continuación podremos ver la estructura del patrón en la figura 5.14.

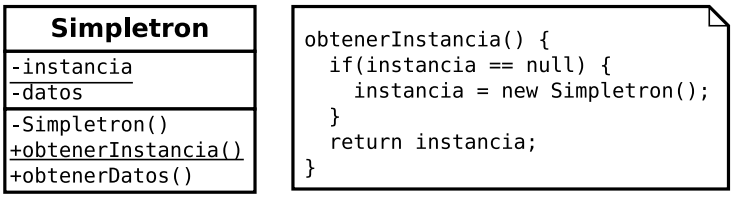


Figura 5.14.: Estructura del patrón Instancia Única

A la clase que cumple el patrón de instancia única se le suele llamar "simpletrón". En este caso, la clase tiene como atributo una instancia de sí misma estática, a la cuál se puede acceder mediante el método `obtenerInstancia`. Este método, si la instancia no está creada la crea, primero y la devuelve luego. Nótese que el constructor de esta clase es privado. Esto evita la instanciación de objetos de *Simpletron* desde el exterior, ayudando al cumplimiento del patrón.

Visitante El patrón Visitante (*Visitor*) [23] es un patrón de comportamiento empleado a la hora de aplicar una operación externa a un conjunto de clases ya existentes, sin modificar su comportamiento.

En nuestros bloques, este patrón servirá para recorrer las estructuras de datos que representan tanto a la composición como a la especificación, y nos permitirá recopilar las constantes y variables existentes. La estructura del patrón está disponible en la figura 5.15.

Como podemos ver, el cliente, para realizar la visita a los objetos que implementen la interfaz *Elemento*, ejecuta en ellos el método `aceptar`, el cual indica el visitante que desea recibir.

El trabajo del visitante es realizar la operación que desea realizar por cada *Elemento*. Normalmente, este patrón se aplica en elementos compuestos, y tras realizar el trabajo, el elemento aceptaría al visitante para cada uno de sus componentes hasta llegar a una hoja.

5.5.2.2. Sistema de tipos de constantes

Para poder averiguar qué variables de una especificación TestSpec son compatibles con las constantes, debemos de asignarles a las mismas un

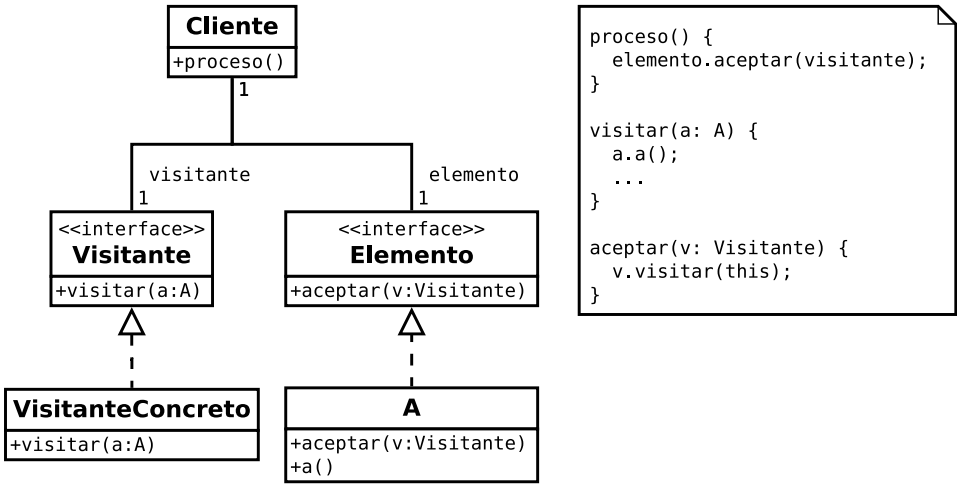


Figura 5.15.: Estructura del patrón Visitante

tipo a la hora de crearlas. Para ello, aprovechamos el sistema de tipos básicos creado para el lenguaje TestSpec [40].

Por lo tanto, tendremos constantes de tipo `string`, `int`, `float`, `date`, `time`, `dateTime` y `duration`.

Como consecuencia, el bloque secundario de gestión de constantes implementará este sistema de tipos.

5.6. Implementación

En esta sección se detallará la estructura interna de cada uno de los subsistemas que forman parte de TestGenerator-Autoseed, así como los detalles de implementación de la misma.

Cada uno de los diagramas de clases Java que veremos a continuación contienen los atributos y operaciones clave para entender el funcionamiento de los bloques. Por el mismo motivo, se excluyen las clases de utilidades existentes de los mismos, dado que se utilizan para descargar de responsabilidad a las clases encargadas de realizar la funcionalidad requerida. El código Java que implementa los bloques está disponible en el CD adjunto y en el repositorio de código del grupo UCASE [30].

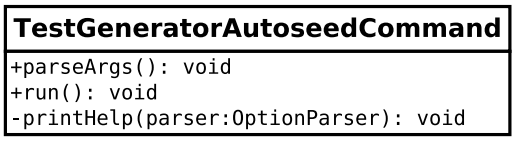


Figura 5.16.: Diagrama de clases Java del subsistema de interacción con el usuario

5.6.1. Subsistema de interacción con el usuario

El bloque de interacción con el usuario tiene como objetivo el de capturar la petición que el usuario desea de la herramienta, comprobar la corrección de la misma y de los parámetros necesarios para procesarla y enviarla al bloque de control para su procesamiento.

En la figura 5.16 podemos ver el diagrama de clases Java correspondiente a este bloque.

En él vemos la clase que realiza la tarea de CLI, *TestGeneratorAutoseedCommand*. Esta clase se encarga de procesar la petición del usuario a través de la línea de órdenes, comprobando que la orden y los parámetros necesarios para su ejecución sean los esperados (método `parseArgs`). Para ello nos apoyamos en la librería JOptSimple [44].

Una vez procesada, delega en el subsistema de control la realización de las acciones pertinentes (método `run`), salvo en el caso de petición de ayuda, donde es la propia clase la que se encarga de mostrar la misma al usuario mediante el método `printHelp`.

5.6.2. Subsistema de control

Este bloque encapsula al algoritmo de aplicación de la técnica de siembra automática. Tiene como tarea la de realizar las peticiones adecuadas al resto de subsistemas principales para llevar a cabo la petición recibida desde el bloque de interacción con el usuario.

En la figura 5.17 podemos ver el diagrama de clases Java correspondiente a este bloque.

La clase encargada del control, y que por tanto encapsula al algoritmo de siembra automática es *AutomaticSeeding*. Este algoritmo, el cual se

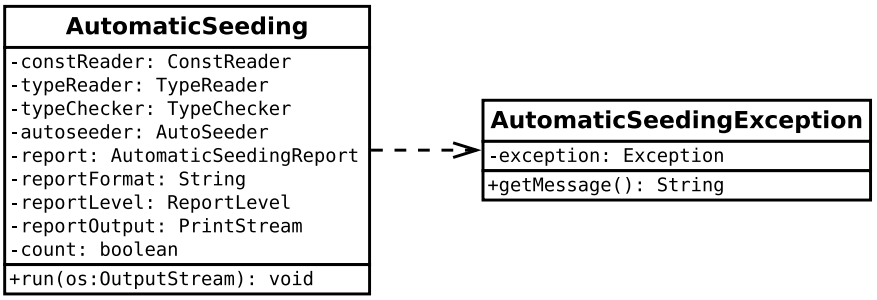


Figura 5.17.: Diagrama de clases Java del subsistema de control

ejecuta a través del método `run`, delega las tareas secundarias en los subsistemas especificados en 5.5.1.

En el caso de que ocurriese algún fallo a la hora de aplicar la técnica, la excepción pertinente se encapsularía en un objeto de la clase *AutomaticSeedingException*, haciendo más fácil la gestión de errores.

5.6.3. Subsistema de lectura

Este bloque es el encargado de realizar la lectura de los elementos que el algoritmo de siembra automática necesita para su ejecución. Dado que el algoritmo necesita leer tanto constantes como variables, este bloque se subdivide en dos bloques, los cuales son:

- Bloque de lectura de constantes (*constants*).
- Bloque de lectura de variables de la especificación (*variables*).

A continuación veremos cuál es la funcionalidad de ambos bloques y de qué están compuestos los mismos.

5.6.3.1. Bloque de lectura de constantes

El bloque de lectura de constantes es el encargado de obtener las constantes existentes en una composición WS-BPEL, además de las variables relacionadas a las mismas.

En la figura 5.18 podemos ver el diagrama de clases Java correspondiente a este bloque.

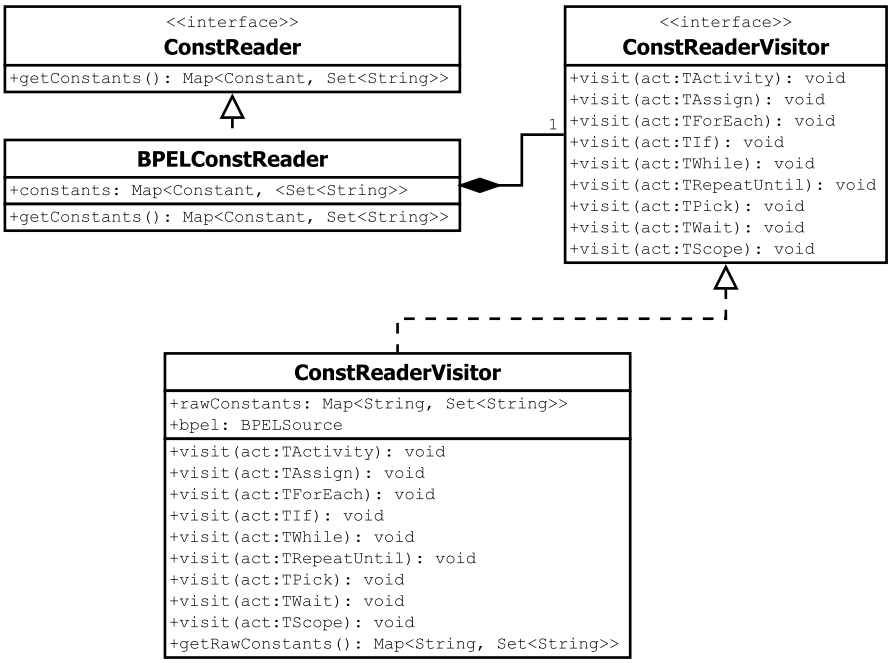


Figura 5.18.: Diagrama de clases Java del bloque de lectura de constantes

En primer lugar, este bloque ofrece la interfaz *ConstReader*, fachada del subsistema, la cual contiene el contrato necesario que cualquier lector de constantes debe cumplir, siendo en este caso el método `getConstants`.

La existencia de esta interfaz permitirá en el futuro desarrollar nuevas clases que podrían realizar la búsqueda de constantes en un origen diferente a una composición WS-BPEL, por ejemplo un programa en Java o C++.

Nuestro lector de constantes es un objeto de clase *BPELConstReader*, el cual implementa la fachada anterior. El proceso de creación de constantes es posterior a su lectura, así pues, se necesita del recorrido apropiado de la composición WS-BPEL para obtener las constantes sin tipo asignado. Para ello utilizamos una variante del patrón de diseño visitante 5.5.2.1, en el sentido de que la clase que juega el rol de elemento visitado no puede ser modificada debido a su generación automática a través de XMLBeans [19].

Esta variante del patrón se ve reflejada en la interfaz *ConstReaderVisitor* y en la clase que la implementa, *BPELConstReaderVisitor*. Sobre la clase visitante recae la responsabilidad de obtener las constantes y las variables de la composición relacionadas.

5.6.3.2. Bloque de lectura de variables

El bloque de lectura de variables tiene como labor la de obtener las variables existentes en una especificación TestSpec.

En la figura 5.19 podemos ver el diagrama de clases Java correspondiente a este bloque.

El diseño en este caso es análogo al del bloque de lectura de constantes (ver 5.6.3.1). Se ofrece la interfaz *TypeReader*, la cual permite sustituir cómodamente la implementación actual por otra diferente en caso de ampliaciones futuras.

La clase encargada de leer las variables es *SpecReader*. Para ello, se utiliza el parser de TestSpec creado para la herramienta TestGenerator [40]. Para extraer las variables de tipo básico existentes dentro de tuplas y listas, se implementa la interfaz externa *ITypeVisitor*, correspondiente a TestGenerator, la cual forma parte de un patrón visitante, como en el caso anterior. La clase encargada de ello es *TypeVisitor*.

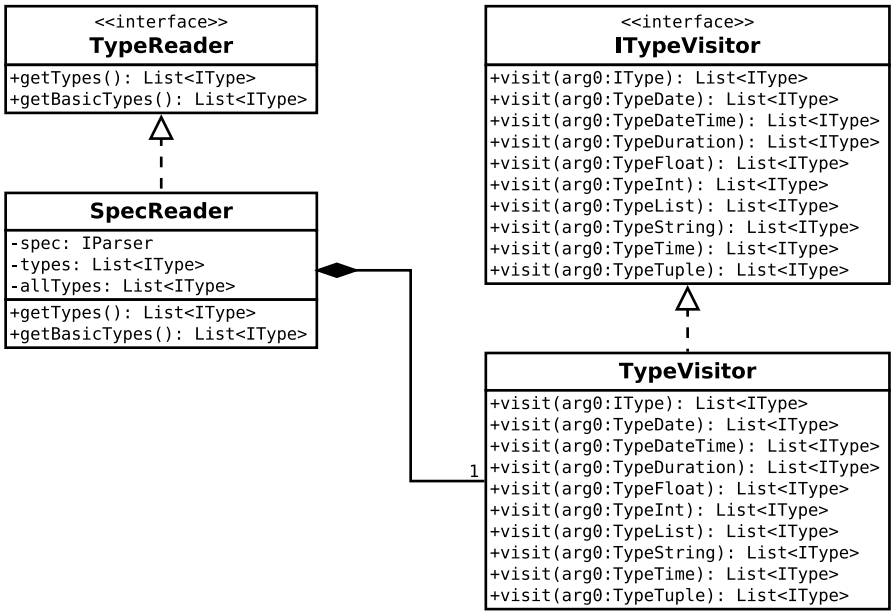


Figura 5.19.: Diagrama de clases Java del bloque de lectura de variables

5.6.4. Subsistema de chequeo

Este bloque tiene como tarea principal la de encontrar, para cada variable de la especificación, el conjunto de constantes compatibles con ella.

Para comprobar si existe compatibilidad entre una constante y una variable de la especificación dadas, se realizarán los siguientes pasos:

1. Comprobación de compatibilidad de tipos entre ambas.
2. Verificación de cumplimiento de las restricciones propias de la variable.
3. En el caso de activarse la optimización, verificación de las restricciones que ésta aporta.

Teniendo en cuenta lo anterior, este subsistema se dividirá en los siguientes bloques:

- Bloque de control (*algorithm*).
- Bloque de estrategias (*strategies*).

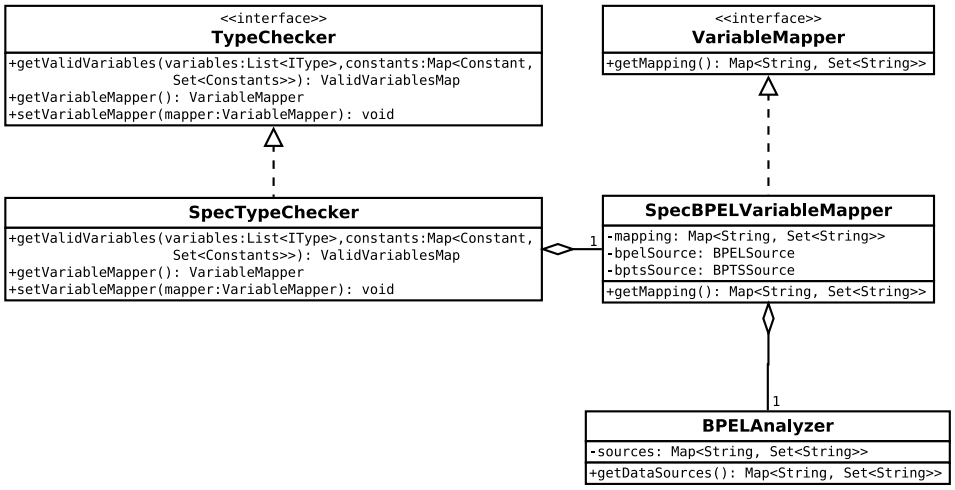


Figura 5.20.: Diagrama de clases Java del bloque de control del subsistema de chequeo

En las siguientes secciones se analizará la funcionalidad de cada uno de los subbloques y se comentará su composición.

5.6.4.1. Bloque de control

El bloque de control tiene como misión la de llevar a cabo las comprobaciones de compatibilidad establecidas anteriormente. Podemos ver en la figura 5.20 el diagrama de clases Java correspondiente a este bloque.

Al igual que en los otros subsistemas principales, tenemos una interfaz, *TypeChecker*, la cual hace de fachada hacia el subsistema. La clase encargada del chequeo de la compatibilidad es *SpecTypeChecker*, obteniéndose las variables junto al conjunto de constantes compatibles mediante el método `getValidVariables`.

Debido a la cantidad de posibilidades de chequeo que pueden darse, esta clase delega la responsabilidad de seleccionar el algoritmo apropiado de chequeo en el subsistema de estrategias. En cuanto a la optimización, la clase que se encarga de realizar el mapeo entre variables de la especificación TestSpec y variables de la composición WS-BPEL es *SpecBPELVariableMapper*.

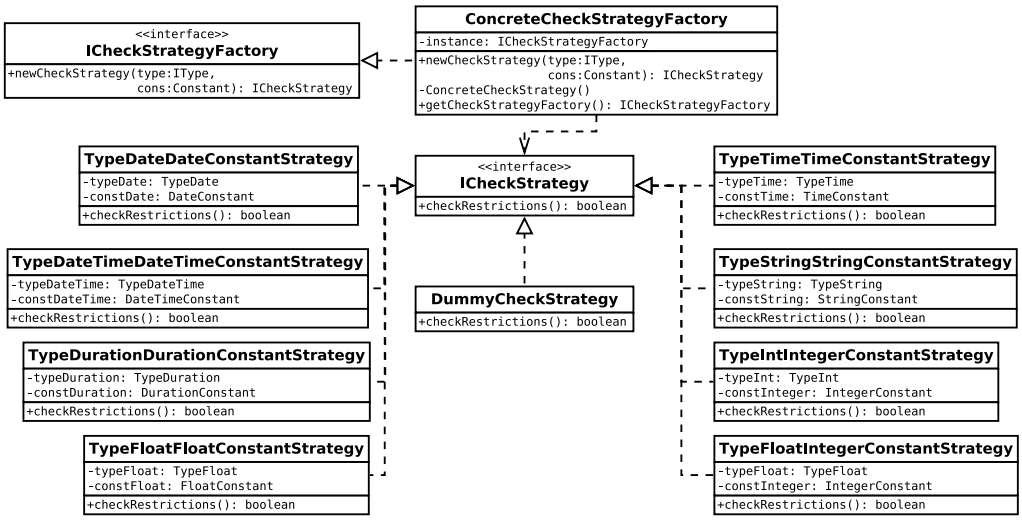


Figura 5.21.: Diagrama de clases Java del bloque de estrategias

Esta clase, que implementa la interfaz mínima *VariableMapper* que debe tener una clase encargada de su tarea, permite obtener el mapeo a través de su método `getMapping`. La responsabilidad de la búsqueda del origen de las variables de la composición recae sobre la clase *BPELAnalyzer*. Esta clase, mediante su método `getDataSources`, permite encontrar los orígenes de las variables de la composición WS-BPEL que se puedan localizar siguiendo una cadena de asignaciones.

5.6.4.2. Bloque de estrategias

La tarea del bloque de estrategias es la de obtener el algoritmo de chequeo de restricciones adecuado en función del tipo de constante y de variable dados. En la figura 5.21 podemos ver el diagrama de clases Java correspondiente a este bloque.

Vemos la aplicación de los patrones de diseño Método Fábrica (ver 5.5.2.1) y Estrategia (ver 5.5.2.1) al mismo tiempo. La clase *SpecTypeChecker*, del bloque anterior, depende de la interfaz *ICheckStrategy*. Debido a que el tipo de estrategia se decide en función de los tipos de la constante y la variable, la estrategia delega en la fábrica de interfaz *ICheckStrategyFactory* el proceso de creación de estrategias.

La nomenclatura de las estrategias se debe al uso del mecanismo de introspección del lenguaje Java [34], mediante el cual la fábrica selecciona la estrategia a crear en función del nombre de las clases de los objetos que recibe. Esto permite una enorme flexibilidad a la hora de registrar nuevas estrategias.

5.6.5. Subsistema de generación de informes

Este bloque es el encargado de la generación del informe relativo al proceso de siembra automática realizado. Los atributos que posee el informe son los siguientes:

- Nivel de detalle. Se definen 3 niveles de detalle jerárquicos para el informe, que son:

  Básico Incluye información sobre las variables de la especificación encontradas, así como su tipo y constantes compatibles con ellas.

  Avanzado Incluye además información sobre las constantes encontradas en la composición, variables relacionadas con ellas y variables de la especificación encontradas.

  Completo Incluye información sobre el mapeo establecido entre las variables de la especificación y las de la composición, en caso de haber activado la optimización.

- Formato de salida. Se provee una estructura que permite formatear el informe de manera cómoda. Además, se provee soporte para obtener el informe en formato de texto plano, XML y YAML.

A continuación veremos el diagrama de clases Java que conforma este subsistema, en la figura 5.22.

Tenemos la clase *Report*, encargada de almacenar la información que se ha ido generando durante el proceso de siembra automática. Para desacoplar el informe del formato que tenga, creamos la interfaz *IReportFormatter*, asignando a las clases que la implementen la tarea de formatear el informe.

Debido a la existencia de diferentes formatos, aplicamos de nuevo el patrón Método Fábrica sumado a la introspección de Java, al igual que se hizo en el bloque 5.6.4.2. En este caso, la fábrica, la cual respeta como

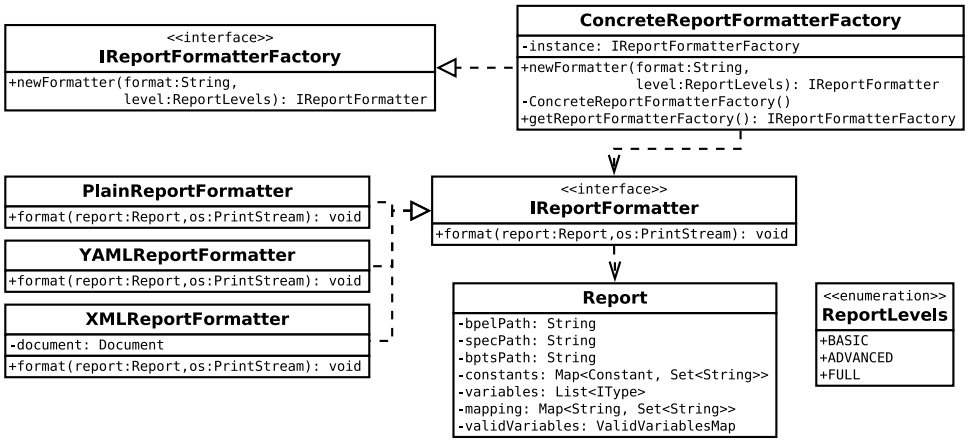


Figura 5.22.: Diagrama de clases Java del bloque de generación de informes

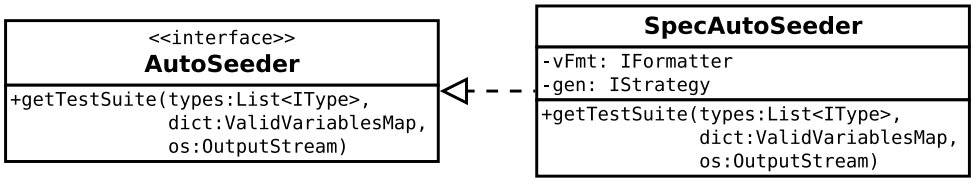


Figura 5.23.: Diagrama de clases Java del bloque de generación de casos de prueba

en el resto de ocasiones el patrón de Instancia Única (Ver 5.5.2.1), selecciona el formato a aplicar en función del nombre de formato recibido.

5.6.6. Subsistema de generación de casos de prueba

Este bloque tiene la responsabilidad de, una vez halladas las variables de la especificación y las constantes compatibles con ellas, generar un conjunto de casos de prueba aleatorio base y realizar las modificaciones pertinentes que la técnica define. En la figura 5.23 está disponible el diagrama de clases Java que implementa al bloque.

Se ofrece la interfaz *AutoSeeder*, como fachada del subsistema. La clase *SpecAutoSeeder* implementa la interfaz y delega la generación del conjunto de casos de prueba aleatorio al generador de TestGenerator. Debido a

la reestructuración de TestGenerator en torno a estrategias, se ha tomado la estrategia de generación que emplea una distribución uniforme. En el futuro se añadirá la posibilidad de cambiar al resto de estrategias disponibles en TestGenerator.

También se delega la responsabilidad de formatear el conjunto de salida a las clases pertinentes de TestGenerator. En este caso, elegimos el formateador de salida de Apache Velocity. Por lo tanto, la responsabilidad de la clases es la de realizar las modificaciones pertinentes en el conjunto de casos de prueba generado para aplicar la técnica.

5.6.7. Bloques secundarios

A continuación mostraremos la funcionalidad común que aglutina cada uno de los bloques secundarios de nuestra herramienta.

5.6.7.1. Bloque de gestión de fuentes

Este bloque contiene una serie de clases que se encargan de encapsular las diferentes fuentes de datos, en este caso, los ficheros de la composición WS-BPEL y del conjunto de casos de prueba BPELUnit. Dado que ambas fuentes están descritas mediante XML, se añade una clase madre *XMLSource* con funcionalidad común. Todo esto se puede apreciar en la figura 5.24.

Es importante destacar que la clase *BPELSource* utiliza la herramienta ServiceAnalyzer [26] para generar el catálogo de servicios web involucrados en la composición. Este catálogo se emplea a la hora de aplicar la optimización, para comparar el formato de mensaje con el de las plantillas que están en la clase *BPTSSource*.

5.6.7.2. Bloque de gestión de constantes

Este bloque auxiliar contiene el sistema de tipos de constantes empleado por los diferentes subsistemas que conforman TestGenerator-Autoseed. Para crear las diferentes constantes en función de la constante que se le pase como cadena, se aplica de nuevo el patrón Método Fábrica, el cual clasifica la constante en torno a expresiones regulares. El diagrama de clases correspondiente a este bloque está en la figura 5.25.

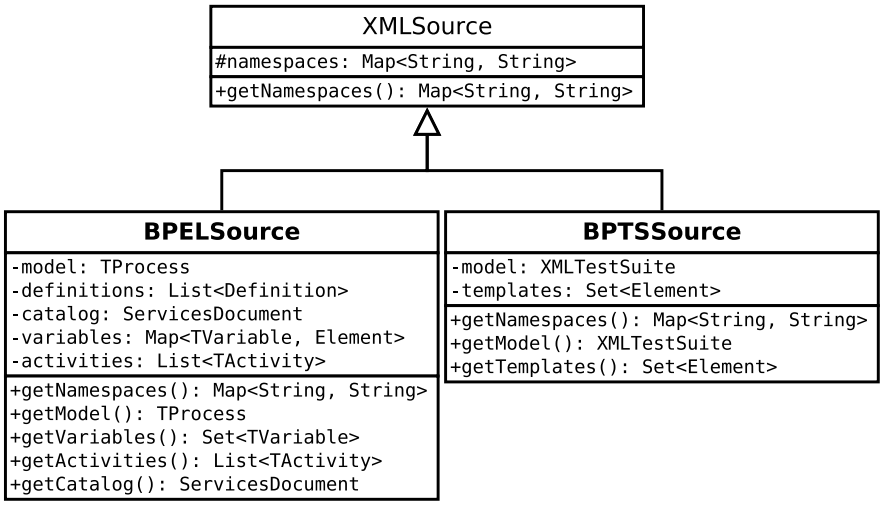


Figura 5.24.: Diagrama de clases Java del bloque de gestión de fuentes

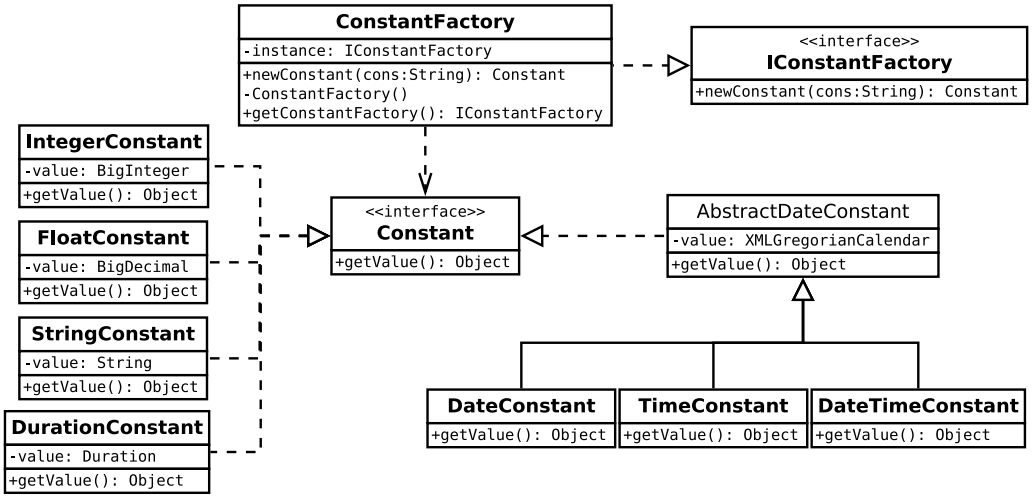


Figura 5.25.: Diagrama de clases Java del bloque de gestión de constantes

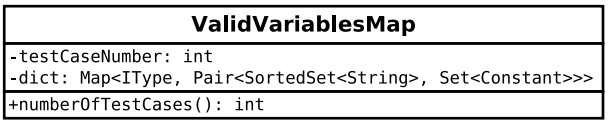


Figura 5.26.: Diagrama de clases Java del bloque de utilidades

5.6.7.3. Bloque de utilidades

Este bloque contiene la clase que posee la estructura que almacena las variables junto a las constantes compatibles con ellas, en la clase *ValidVariablesMap*, encargada de calcular el número de casos de prueba necesarios para la técnica. Debido que incluir esta clase en cualquier subsistema aumentaba el acoplamiento entre ellos, se decidió sacar a uno externo para evitar este hecho. Podemos ver esta clase en la figura 5.26.

5.7. Pruebas

En esta sección se detalla la fase de pruebas correspondiente al desarrollo de la herramienta TestGenerator-Autoseed.

5.7.1. Plan de pruebas

5.7.1.1. Alcance

Se han realizado pruebas unitarias a cada uno de los subsistemas por separado y pruebas de integración de los subsistemas que conforman el subsistema de control.

5.7.1.2. Tiempo y lugar

Las pruebas se han llevado a cabo mientras se desarrollaba el proyecto.

Una vez culminada la fase de implementación de cada una de las versiones de la herramienta, se ejecutaban las pruebas unitarias existentes y se añadían pruebas unitarias nuevas correspondientes a la funcionalidad añadida. En caso de tener éxito, se cerraba la versión y se pasaba al

análisis de los nuevos requisitos pedidos. En caso contrario, se procedía a depurar los errores y se volvían a ejecutar las pruebas.

Una vez finalizado el añadido de requisitos, en la versión final se realizaron pruebas de integración del sistema completo, con el objeto de comprobar que no existían errores en el mismo.

5.7.1.3. Naturaleza

Todas las pruebas realizadas han sido en torno a la especificación del sistema (de caja negra). Recordemos que las pruebas de caja negra se centran en verificar el dominio de las entradas y salidas del software a probar, al contrario que las pruebas de caja blanca, que se encargan de comprobar que la estructura interna del software es la adecuada.

5.7.2. Diseño de las pruebas

Para crear las pruebas, se ha empleado el framework JUnit [27]. Este framework aporta una serie de clases que sirven para implementar de manera cómoda el conjunto de casos de prueba necesario empleando el lenguaje Java. Las clases Java que implementan las pruebas están disponibles en el CD adjunto y en el repositorio del proyecto [30].

Dada la arquitectura de nuestro sistema, las pruebas unitarias se han orientado hacia los siguientes bloques:

Subsistema de lectura de constantes Para probar este subsistema, se necesita de un conjunto de composiciones WS-BPEL lo suficientemente diferentes para que podamos encontrar los diferentes tipos de constantes existentes. Por lo tanto, creamos una clase abstracta madre, *GenericConstReaderTest*, de la cual heredan las clases de prueba para cada composición del repositorio. Los parámetros a probar han sido el número y tipo de constantes obtenidas tras la lectura.

Subsistema de lectura de variables En este caso únicamente necesitamos probar el hecho de hallar correctamente todas las variables de tipo básico, estén o no dentro de una variable de tipo complejo, ya que la prueba de la lectura de ficheros TestSpec se realiza en TestGenerator.

Subsistema de chequeo En este caso necesitamos comprobar que la estrategia de chequeo apropiada se selecciona correctamente (*CheckStrategyTest*) y que el chequeo de compatibilidad de variables en sí se hace de forma correcta. Al igual que en el subsistema de lectura de constantes, se realiza la prueba con composiciones WS-BPEL, utilizando esta vez además sus especificaciones TestSpec.

Subsistema de generación del informe La prueba de este subsistema gira en torno a verificar que se realiza la generación de forma correcta. Para ello, se verifica que la selección del formato y del nivel de detalle originan el informe que se desea.

Subsistema de generación de casos de prueba En este subsistema es necesario verificar que la modificación que la técnica de la siembra automática realiza al conjunto de casos de prueba base es correcta.

Subsistema de interacción con el usuario En este caso las pruebas deben verificar que el procesamiento de la petición realizada por el usuario se maneja correctamente, tanto para casos de éxito como de error.

Debido a que en nuestro caso las pruebas de la CLI llevan consigo la integración con el subsistema de control, las pruebas de integración han estado orientadas únicamente al subsistema de control, el cual se encarga de integrar al resto de bloques que realizan la funcionalidad requerida.

Las composiciones WS-BPEL para llevar a cabo dichas pruebas deben de ser lo suficientemente diferentes para que así las pruebas sean significativas. Para ello, se escogieron las siguientes composiciones del repositorio de composiciones del grupo:

- `LoanApprovalDoc`
- `LoanApprovalExtended`
- `MarketPlace`
- `ShippingSync`

La elección de estas composiciones se debe a su diferencia en cuanto a tamaño, número de constantes que contienen, cantidad de variables que se logran mapear en ellas y efectos de la optimización de la técnica.

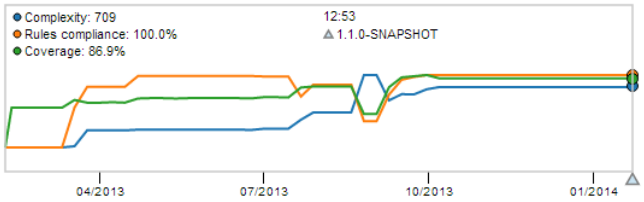


Figura 5.27.: Evolución de este PFC en Sonar

5.7.3. Validación

Para comprobar que el código fuente que implementa la funcionalidad requerida se ajusta a unos criterios adecuados de calidad, empleamos la herramienta Sonar, como se especificó en 4.4.4.

En la figura 5.27 podemos ver una gráfica en la que se puede ver la evolución a lo largo del tiempo de la complejidad, el porcentaje de cobertura de las pruebas y el porcentaje de cumplimiento de las reglas de programación en Java.

Los saltos en la línea de complejidad nos permiten ver los momentos en los que se añadió nueva funcionalidad a la herramienta. Tiene un máximo, coincidente con el momento de inclusión de la optimización. El posterior rediseño permitió bajar el nivel de complejidad, haciendo más simple mantener el código.

El máximo en la complejidad implicó un mínimo tanto en el cumplimiento de las reglas de programación como en la cobertura de las pruebas. El rediseño y la inclusión de pruebas elevaron estos niveles hacia los actuales.

Esta herramienta permitió además realizar el desacoplamiento de los distintos paquetes que conforman los subsistemas de la herramienta, eliminando algunos ciclos de dependencias existentes.

6. Estudio comparativo entre la técnica de generación aleatoria y la siembra automática

En este capítulo se realizará un estudio comparativo de la calidad de los conjuntos de casos de prueba generados mediante la técnica aleatoria, la técnica de siembra automática y la técnica de siembra automática optimizada, comparando los resultados obtenidos.

6.1. Selección de composiciones a estudiar

En primer lugar, es necesario seleccionar una serie de composiciones WS-BPEL de las que el grupo UCASE dispone en su repositorio [49]. El criterio de selección empleado es el número de casos de prueba necesario para aplicar la siembra automática, tanto en su versión convencional como optimizada.

Para ello, empleamos la herramienta TestGenerator-Autoseed, utilizando su capacidad de contar el número de casos de prueba necesarios para aplicar la técnica. En la tabla 6.1 podemos ver los resultados de esta operación.

A partir de estos resultados podemos extraer una serie de conclusiones, que son las siguientes:

- Tenemos un conjunto de composiciones, $\{1, 5, 6\}$, donde la optimización no tiene efecto alguno. Esto se debe que todas las constantes existentes en ellas tienen relación con las variables de sus correspondientes especificaciones TestSpec.

	Composición	Convencional	Optimizada
1	`LoanApprovalDoc`	5	5
2	`LoanApprovalExtended`	561	69
3	`LoanApprovalExtendedParallel`	592	76
4	`LoanApprovalExtendedWoutAvls`	274	38
5	`LoanApprovalPaths`	5	5
6	`LoanApprovalRPC`	3	3
7	`MetaSearch`	216	3
8	`ShippingSync`	7	0
9	`SquaresSum1`	2	0
10	`SquaresSum2`	2	0
11	`SquaresSum3`	3	2
12	`TradeIncome`	41	3
13	`TravelReservationService`	902	0

Tabla 6.1.: Resultados de contar el número de casos de prueba por cada composición

- Por otra parte tenemos el conjunto $\{7, 8, 9, 10, 12, 13\}$, composiciones en las que la optimización no es lo suficientemente eficaz en términos de encontrar las relaciones existentes entre variables de la composición y de la especificación. Esto puede ser debido a diferentes motivos:
  - Imposibilidad de generar el catálogo de servicios web mediante ServiceAnalyzer. En este caso, la optimización no puede aplicarse, ya que se necesita este catálogo en el proceso. Ocurre en la composición `TravelReservationService` (13).
  - Plantilla de mensaje dentro del conjunto de casos de prueba BPELUnit no adecuada. Debido a esto, la comparación realizada en el proceso de optimización es errónea, y se diluyen las relaciones existentes. Es el caso de `ShippingSync` (8).
  - Flujo de datos complejo. En el resto de composiciones del conjunto, el flujo de datos es lo suficientemente complejo como para que el algoritmo basado en asignaciones no encuentre el origen de una variable.
- El resto de composiciones están en los valores esperados a la hora de aplicar la optimización de la técnica.

Vistos los resultados, seleccionamos las composiciones `LoanApproval-Doc` y `LoanApprovalExtendedWoutAvls` para realizar los estudios de calidad. Se descartan el resto de composiciones seleccionables dada la similitud existente con alguna de las dos escogidas.

6.2. Proceso a realizar

Una vez tenemos las composiciones WS-BPEL seleccionadas, procedemos a realizar el estudio en sí mismo. Para ello, se realiza lo siguiente para cada una de las composiciones:

1. Se generan los conjuntos de casos de prueba empleando la técnica de generación aleatoria y la siembra automática convencional y optimizada. Para que los resultados sean significativos, los valores aleatorios en todos los casos deben ser los mismos, ya que valores diferentes distorsionarían los resultados obtenidos. Además, si se comparan la técnica aleatoria con la siembra automática, el tamaño de ambos conjuntos debe coincidir.

2. Se realiza el estudio de prueba de mutaciones empleando la herramienta MuBPEL [48]. Para ello es necesario:

   a) Generar los mutantes correspondientes a la composición.

   b) Ejecutar la composición original frente al conjunto de casos de prueba con los valores generados en cada caso.

   c) Comparar la salida de los mutantes con la de la composición original y obtener la puntuación de mutación, que nos indica la calidad del conjunto de casos de prueba.

Una vez obtenidas ambas puntuaciones de mutación, se comparan y se extraen las conclusiones pertinentes del estudio. Especialmente en el caso del estudio de la siembra automática optimizada, es conveniente comparar el tiempo invertido en cada caso.

Con el objetivo de mejorar la resolución del estudio, dado que utilizamos orígenes aleatorios, es conveniente utilizar más de un punto de partida aleatorio para obtener resultados adecuados. En este caso, debido al coste de la ejecución de los casos de prueba, se ha limitado el estudio a 3 rondas en el caso de `LoanApprovalDoc` y 2 rondas para `LoanApprovalExtendedWoutAvls`.

6.2.1. Restricciones

Hay que tener en cuenta una consideración a la hora de realizar estudios de prueba de mutaciones utilizando conjuntos de casos de prueba generados mediante técnicas aleatorias o derivadas como la siembra automática. Tenemos que evitar el uso de plantillas en las que las variables de la especificación tienen dependencias entre ellas, como veremos en el siguiente ejemplo.

```
typedef date (min="2010-01-01", max="2014-01-01")
  FechaInicio;
typedef int (min=1, max=180) DiasProyecto;
typedef int (min=1, max=7) DiasVacaciones;
typedef tuple(element=DiasProyecto, DiasVacaciones)
  Proyecto;
typedef list (element=Proyecto, min=1, max=5)
  Proyectos;
```

```
typedef tuple (element=FechaInicio, Proyectos)
  InfoProyectos;
InfoProyectos proyectos;
```

En esta especificación TestSpec se pretende generar un conjunto de casos de prueba de una serie de proyectos, en los que hay que generar la fecha de inicio, el número de días de duración y el número de días de vacaciones al terminar el proyecto.

Supongamos que en la composición se realiza el cálculo del número de días trabajados en todos los proyectos hasta el día de hoy. Si realizamos un caso de prueba que verifique la corrección de este proceso de cálculo y generamos un conjunto de datos aleatorios para ello, existe una alta probabilidad de que el número de días por proyecto sea superior al número de días que realmente han transcurrido entre la fecha de inicio y la de hoy.

Esto hace que el caso de prueba falle, y es una fuente de errores muy difícil de detectar. Por lo tanto, es necesario evitar en las plantillas este tipo de dependencias, o bien, si son inevitables, calcular los límites de forma adecuada para que las dependencias se satisfagan independientemente de los valores generados.

Dentro de esta categoría se enmarcaría también el uso de plantillas de casos de prueba “inteligentes”, en las que los mockups responden en función de ciertos parámetros de la entrada. Este hecho es una dependencia de variables disfrazada, ya que obliga a correlacionar de algún modo la respuesta del mockup con la entrada generada, algo que no es posible en técnicas aleatorias.

Debido a ello, TestGenerator incorpora además una serie de estrategias no aleatorias, basadas en este tipo de restricciones. En este estudio no son utilizadas, aunque se prevé su uso en estudios posteriores.

6.3. Estudio de impacto de la optimización en la siembra automática

En este apartado se presentan los resultados obtenidos tras realizar el proceso descrito en la sección 6.2 empleando conjuntos de casos de

Datos generales					
Número de mutantes generados					4500
Convencional					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	77,55 %	22,38 %	0,07 %	22,45 %	5d 3h 55m 40s
2	78,62 %	21,31 %	0,07 %	21,38 %	5d 2h 24m 59s
Media	78,08 %	21,85 %	0,07 %	**21,92 %**	**5d 3h 10m 19s**
Número de casos de prueba					274
Optimizada					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	80,51 %	19,42 %	0,07 %	19,49 %	17h 52m 3s
2	79,97 %	19,96 %	0,07 %	20,02 %	16h 19m 19s
Media	80,24 %	19,69 %	0,07 %	**19,76 %**	**17h 5m 41s**
Número de casos de prueba					38

Tabla 6.2.: Resultados del estudio de impacto de la optimización

prueba generados mediante siembra automática convencional y optimizada.

Este estudio se realiza únicamente para la composición `LoanApproval-ExtendedWoutAvls`, ya que para la composición `LoanApprovalDoc` la optimización no tiene efecto. En la tabla 6.2 podemos ver los resultados obtenidos tras realizar el estudio.

Para esta composición, el impacto de la optimización supone un descenso aproximado del **2,16 %** en la puntuación de mutación obtenida. Si observamos los tiempos medios obtenidos, podemos comprobar que obtenemos un **86,13 %** de reducción respecto a la ejecución de la técnica de siembra automática convencional.

Datos generales					
Número de mutantes generados					98
Número de casos de prueba					5
Aleatoria					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	33,67 %	64,29 %	2,04 %	66,33 %	1m 48s
2	65,30 %	32,65 %	2,04 %	34,70 %	1m 50s
3	65,30 %	32,65 %	2,04 %	34,70 %	1m 55s
Media	54,76 %	43,20 %	2,04 %	**45,24 %**	**1m 51s**
Siembra					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	31,63 %	66,33 %	2,04 %	68,37 %	55s
2	10,20 %	87,76 %	2,04 %	89,80 %	5m 29s
3	19,39 %	78,57 %	2,04 %	80,61 %	6m 38s
Media	20,41 %	77,55 %	2,04 %	**79,59 %**	**4m 57s**

Tabla 6.3.: Resultados del estudio para `LoanApprovalDoc`

6.4. Estudio comparativo entre la generación aleatoria y la siembra automática

En este apartado podremos ver los resultados que se han obtenido al haber realizado el proceso descrito en la sección 6.2 empleando en esta ocasión conjuntos de prueba generados mediante la técnica aleatoria y siembra automática.

En el caso de la composición `LoanApprovalDoc`, se han obtenido los resultados disponibles en la tabla 6.3.

En negrita tenemos los valores de puntuación de mutación medios obtenidos tras realizar el estudio. Como podemos ver, para estos tres ejemplos se obtiene una mejora aproximada del **34,35 %** aplicando la técnica de siembra automática.

En cuanto a los tiempos, aplicando la siembra automática se obtiene un

aumento de tiempo aproximado del **168 %** frente al tiempo obtenido aplicando la técnica aleatoria. Cabe destacar que en ambos casos el número de casos de prueba ejecutados es el mismo, por lo que la diferencia de tiempo puede ser debida a que los mutantes mueren por agotamiento del tiempo de ejecución establecido para los mismos.

En el caso de la composición `LoanApprovalExtendedWoutAvls` tenemos dos resultados, provenientes de la comparativa entre la técnica aleatoria y la siembra automática convencional, por una parte, y la técnica aleatoria y la siembra automática optimizada por otra. Ambos están disponibles en la tabla 6.4.

Atendiendo al caso de comparar la técnica aleatoria y la siembra automática convencional, podemos ver que se obtiene una leve mejora del **0,02 %** y un aumento de tiempo del **1,7 %** a la hora de aplicar la segunda técnica. En el caso de la comparativa con la siembra automática optimizada, conseguimos una mejora del **0,09 %** y tiempos idénticos (**0 %**).

6.5. Conclusiones

Analizando los datos que hemos obtenido tras realizar los estudios disponibles en 6.3 y 6.4, podemos extraer una serie de conclusiones que veremos a continuación.

En cuanto al impacto de la optimización a la hora de aplicar la siembra automática, hemos obtenido un descenso del tiempo de ejecución muy grande a costa de perder una cantidad ínfima de puntuación de mutación en el conjunto. Esto quiere decir que la optimización realiza su trabajo de forma adecuada, y por lo tanto es recomendable su utilización en casos donde el número de casos de prueba sea muy elevado.

En el caso de la comparativa entre la técnica aleatoria y la siembra automática, en una composición simple como `LoanApprovalDoc` hemos obtenido un gran margen de mejora en la puntuación de mutación, mientras que en una composición compleja como `LoanApprovalExtendedWoutAvls`, hemos obtenido un margen irrisorio. Esto nos lleva a pensar que la siembra automática obtiene mejores conjuntos de casos de prueba que la técnica aleatoria, y el margen de mejora viene dado por las siguientes circunstancias:

Datos generales					
Número de mutantes generados					4500
Número de casos de prueba convencional					274
Número de casos de prueba optimizada					38
Aleatoria					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	77,71 %	22,22 %	0,07 %	22,29 %	5d 3h 25m 23s
2	78,49 %	21,44 %	0,07 %	21,51 %	4d 22h 47m 28s
Media	78,10 %	21,83 %	0,07 %	**21,90 %**	**5d 1h 6m 25s**
Siembra convencional					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	77,55 %	22,38 %	0,07 %	22,45 %	5d 3h 55m 40s
2	78,62 %	21,31 %	0,07 %	21,38 %	5d 2h 24m 59s
Media	78,08 %	21,85 %	0,07 %	**21,92 %**	**5d 3h 10m 19s**
Aleatoria					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	80,71 %	19,22 %	0,07 %	19,29 %	17h 32m 59s
2	79,95 %	19,98 %	0,07 %	20,04 %	16h 21m 54s
Media	80,33 %	19,60 %	0,07 %	**19,67 %**	**17h 5m 41s**
Siembra optimizada					
Ronda	Vivo	Muerto	Error	PM	Tiempo
1	80,51 %	19,42 %	0,07 %	19,49 %	17h 52m 3s
2	79,97 %	19,96 %	0,07 %	20,02 %	16h 19m 19s
Media	80,24 %	19,69 %	0,07 %	**19,76 %**	**17h 5m 41s**

Tabla 6.4.: Resultados del estudio para `LoanApproval-ExtendedWoutAvls`

- *Relevancia de las constantes dentro de la composición*. En composiciones donde las constantes estén presentes en sentencias condicionales, la siembra automática provee un margen de mejora mucho más amplio que en composiciones donde las constantes estén presentes en asignaciones u otras actividades.
- *Número de casos de prueba obtenidos*. Cuanto mayor es el número de casos de prueba a generar por la siembra automática, más se diluye su efectividad.
- *Dominio de las variables de la especificación*. En especificaciones cuyas variables posean un dominio reducido, la siembra automática pierde efectividad debido a que aumenta la probabilidad de que la técnica aleatoria sea capaz de generar conjuntos de casos de prueba que ejerciten el dominio al completo.

De todas formas, en el futuro es necesario ampliar este estudio con un conjunto de composiciones más rico, con el objetivo de saber en qué medida exacta afectan estas circunstancias.

7. Conclusiones y trabajo futuro

En este capítulo se realizará un resumen de los aspectos clave de este PFC, y se definirán las líneas de trabajo futuro.

7.1. Resumen

Se ha definido una técnica de generación de casos de prueba basada en la técnica aleatoria denominada siembra automática. Debido a la cantidad de casos de prueba que se pueden generar, se ha definido una optimización de la técnica con el objetivo de reducir dicha cantidad. Además, se ha especializado esta técnica para su aplicación en el entorno de las composiciones WS-BPEL.

Una vez definida, se ha implementado una herramienta que aplica esta técnica a partir de una composición, una especificación TestSpec y, en caso de aplicar optimización, una especificación del formato de los casos de prueba para BPELUnit.

A continuación se definió un conjunto de casos de prueba empleando JUnit, con el objetivo de comprobar que la funcionalidad de la herramienta es la esperada.

Tras el proceso de prueba, se ha realizado un estudio empleando las composiciones WS-BPEL disponibles dentro del repositorio del grupo con el objetivo de comprobar el impacto que tiene la optimización a la hora de aplicarla dentro del proceso de siembra automáticay con el objetivo de comparar la aplicación de esta técnica con la generación aleatoria.

Este estudio ha permitido extraer una serie de conclusiones relevantes para conocer en qué condiciones la siembra automática permite obtener mejores resultados.

7.2. Valoración

Este PFC se ha realizado con el objetivo de colaborar en las tareas de investigación del grupo UCASE. Así pues, los resultados obtenidos tras su realización son necesarios para avanzar en la investigación sobre la generación automática de casos de prueba.

Debido a la escasa cantidad de trabajos relacionados con la técnica definida en este PFC y con las mejoras necesarias para obtener mejores resultados de optimización, el tiempo invertido ha sido adecuado.

7.2.1. Objetivos

Los objetivos marcados a la hora de proponer este PFC se han cumplido. Esto significa que ahora el grupo UCASE dispone de una herramienta para generar conjuntos de casos de prueba aplicando la siembra automática y un estudio en el que se puede conocer su efectividad respecto a la técnica aleatoria.

Para llevar a cabo este trabajo, han sido de gran ayuda la documentación generada a la hora de realizar otras herramientas del grupo, como es el caso de:

- TestGenerator [40]
- ServiceAnalyzer [26]
- BPTSGenerator [16]
- MuBPEL [29, 6]

Esta documentación, junto a esta memoria, será de gran ayuda para todo aquel que desee extender la funcionalidad de esta herramienta en el futuro.

7.2.2. Conocimientos adquiridos

La realización de este PFC ha sido muy satisfactoria. El alumno había colaborado con anterioridad en el grupo UCASE, ampliando el conjunto de operadores de MuBPEL desembocando en el PFC de Ingeniería Técnica

en Informática de Sistemas “Operadores de Mutación de Cobertura para WS-BPEL 2.0” [29].

Por lo tanto, el alumno ya tenía conocimientos en el ámbito de la prueba de mutaciones y en la mayoría de tecnologías necesarias para la realización de este PFC. Así pues, el alumno ha ampliado conocimientos en:

- Prueba de mutaciones.
- Generación automática de casos de prueba.
- Servicios web.
- Lenguajes: WS-BPEL, XPath, Java.
- Patrones de diseño de sistemas.
- Frameworks: JUnit, BPELUnit.
- Herramientas: Dia, Eclipse, Subversion, Meld, Apache Maven, Sonar.

Adicionalmente, el alumno ha adquirido una mayor experiencia trabajando en grupo, y ha aprendido una serie de cuestiones, que se pueden resumir en:

- El uso de un SCV es vital en proyectos de mediana y gran entidad, ya que permite compartir el código fuente de forma simple y revertir cambios no deseados que en caso de no disponer del sistema podrían suponer un gran contratiempo a la hora de continuar con el proceso de desarrollo.
- El uso de herramientas como Apache Maven, Jenkins y Sonar ayudan al desarrollador a integrar los cambios que van haciendo en el código fuente comprobando en todo momento la corrección del mismo. En caso de que los cambios vuelvan al proyecto inestable, Jenkins envía un correo a los desarrolladores con los errores encontrados, lo que ayuda a depurarlos de forma más rápida. La herramienta Sonar ayuda en gran medida a realizar un código fácil de mantener y que siga los principios de diseño básicos, pues sus informes indican las violaciones de estos principios y cómo podrían solucionarse.

- En trabajos de investigación, se torna imprescindible el contacto con los tutores, pues el número de trabajos sobre la temática en la que se investiga en este caso es muy escasa, y los correos y asistencia a seminarios permiten al alumno encontrar nuevas ideas que le permiten avanzar en la investigación.

7.3. Trabajo futuro

En la sección 6.1 se realizó una selección de composiciones dentro de las disponibles en el repositorio del grupo UCASE. Los resultados obtenidos han satisfecho con creces las expectativas previas, aunque apuntan a que la optimización no es del todo eficaz en composiciones con un flujo de datos complejo.

Para poder mejorar los resultados en estos casos sería necesario aplicar un algoritmo de análisis de flujo de datos. Debido a la complejidad de su aplicación en WS-BPEL [8] y a que la mayoría de algoritmos [52, 35] requieren de la existencia de un CFG, queda por lo tanto como trabajo futuro el desarrollo de esta estructura para WS-BPEL y la posterior aplicación del algoritmo de flujo de datos. Mediante el empleo del mismo, se espera mejorar los resultados obtenidos para las composiciones con un flujo de datos complejo.

Una mejora adicional de TestGenerator-Autoseed que queda también como trabajo futuro es la de priorizar las variables de la especificación, lo que permitiría generar un subconjunto de casos de prueba cuyo tamaño sería definido por el usuario. Esta mejora está enfocada a evitar que la efectividad de la siembra automática se diluya en composiciones a partir de las cuales se obtiene un gran número de casos de prueba.

Otra mejora a realizar como trabajo futuro consiste en dar soporte a los diferentes tipos de estrategias de generación aleatoria disponibles en TestGenerator. En esta versión de la herramienta sólo es posible generar el conjunto de casos de prueba aleatorio base mediante una estrategia que utiliza la distribución uniforme, mientras que TestGenerator permite estrategias adicionales basadas en otras distribuciones como la normal o Poisson, entre otras.

Por último, es necesario ampliar el estudio realizado en el capítulo 6 incluyendo las mejoras expuestas e incluyendo un conjunto de compo-

siciones más rico, con el objetivo de poder descubrir más características relevantes sobre la técnica de la siembra automática.

8. Agradecimientos

- A mi familia y amigos, por apoyarme y animarme en estos meses de trabajo y esfuerzo.
- A Inmaculada Medina Bulo, por darme la oportunidad de seguir colaborando en el grupo UCASE mediante este proyecto.
- A Antonio García Domínguez y Antonia Estero Botaro, por su dedicación y apoyo.
- Al resto de miembros del grupo UCASE, por su ayuda e ideas a la hora de resolver dudas.

A. Manual de instalación

En este capítulo se detalla cómo instalar TestGenerator-Autoseed sobre una distribución GNU/Linux. En particular, se recomienda instalar la herramienta sobre *Ubuntu* u *openSUSE*, en su versión más reciente.

Instrucciones de instalación

Para instalar TestGenerator-Autoseed, es necesario:

1. Descargar el *script* de instalación en el directorio que se desee desde la URL:
   `https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh`
2. Ejecutar el script de instalación mediante la terminal, como sigue:
   `bash install.sh gamera`
3. Seguir las instrucciones que indique el script para concluir la instalación.

El *script* se encargará de instalar la herramienta TestGenerator-Autoseed y las dependencias de ésta:

- Apache Maven 3.0
- Java 6
- Apache Tomcat 5.5
- XMLBeans 2.3
- BPELUnit 1.6
- JUnit 4.8
- TestGenerator
- ServiceAnalyzer

B. Manual de usuario

A continuación ilustraremos el uso de **TestGenerator-Autoseed**. Para entender la utilidad que posee la herramienta, es importante que el usuario posea los conocimientos descritos en 4.3.

Las instrucciones de instalación de la herramienta se encuentran disponibles en A.

Para ejecutar el programa se introduce en cualquier terminal el comando siguiente:

```
test-generator-autoseed (argumentos)
```

Los argumentos serán explicados en las siguientes secciones. En la mayoría de casos aparecerán los argumentos:

- `bpel`: Fichero de la composición WS-BPEL que deseamos analizar.
- `spec`: Fichero TestSpec con la especificación de las variables necesarias

B.1. Generar un conjunto de casos de prueba

En esta sección se explicará cómo utilizar TestGenerator-Autoseed para generar un conjunto de casos de prueba mediante siembra automática. Para ello, es necesario que ejecutemos en un terminal la siguiente orden:

```
test-generator-autoseed [opciones] bpel spec
```

El resto de parámetros opcionales se explicarán en B.4.

B.2. Contar el número de casos de prueba necesarios

Si queremos contar el número de pruebas que hacen falta para aplicar la siembra automática, es necesario que invoquemos a TestGenerator-Autoseed de la siguiente forma:

```
test-generator-autoseed [opciones] --count bpel spec
```

El parámetro `count` es el encargado de indicar a la herramienta el deseo de realizar la cuenta en vez del proceso de siembra automática.

El resto de parámetros opcionales se explicarán en B.4.

B.3. Obtener ayuda del sistema

Si lo que queremos es obtener la ayuda del sistema, la cual contiene las opciones disponibles y los parámetros necesarios para cada una de ellas, deberemos ejecutar TestGenerator-Autoseed de la siguiente forma:

```
test-generator-autoseed --help
```

La ayuda del sistema también se mostrará en caso de que cometamos algún error especificando los parámetros de la aplicación.

B.4. Argumentos opcionales

A continuación iremos viendo cómo se utilizan los argumentos opcionales de los que dispone la herramienta TestGenerator-Autoseed.

Optimización

Podremos activar la optimización de la técnica de siembra automática empleando la opción `O`, indicando además un fichero `bpts`, con el conjunto de casos de prueba BPELUnit de la misma.

La orden completa sería:

```
test-generator-autoseed -O bpts bpel spec
```

Informe

Podremos personalizar la forma en que se genera el informe empleando las opciones `report-*` que veremos a continuación.

Mediante la opción `report-level` podremos establecer el nivel de detalle del informe a generar, entre los siguientes valores:

- *BASIC*. Únicamente muestra las variables válidas encontradas junto a las constantes compatibles. Es el nivel por defecto cuando no indicamos el nivel de forma explícita.
- *ADVANCED*. Muestra lo anterior más las constantes y variables encontradas.
- *FULL*. Muestra toda la información disponible, incluyendo las relaciones existentes entre las variables de la composición y las de la especificación.

Así quedaría el comando aplicando esta opción:

```
test-generator-autoseed --report-level level bpel spec
```

Siendo `level` uno de los niveles anteriores.

Si utilizamos la opción `report-format`, podremos indicar a la herramienta el formato de salida del informe que queremos obtener, entre uno de los existentes, que son:

- *none*. Empleado para indicar a la herramienta que no queremos generar el informe asociado.

- *plain*. Salida en texto plano, decorada para facilitar su legibilidad. Es el nivel por defecto cuando no indicamos el formato de forma explícita.
- *xml*. Salida en formato XML.
- *yaml*. Salida en formato YAML.

Si elegimos el formato de salida del informe, el comando quedaría de esta forma:

```
test-generator-autoseed --report-format format bpel
spec
```

Siendo `format` uno de los formatos anteriores.

Por último, podemos elegir el fichero destino del informe, con la opción `report-output`. Por defecto, el informe se vuelca a la salida de errores (*System.err*). Si deseamos volcarlo a un fichero en concreto, deberíamos ejecutar:

```
test-generator-autoseed --report-output informe bpel
spec
```

Donde `informe` sería el nombre del fichero que contendrá el informe.

Salida

Por defecto, la herramienta TestGenerator-Autoseed muestra la salida, ya sea el conjunto de casos de prueba en formato VTL o el número de casos de prueba en la consola de la línea de órdenes (*System.out*).

Para modificar este comportamiento, existe la opción `output`, la cual permite indicar el fichero donde queremos volcar la salida. Se utilizaría de la siguiente forma:

```
test-generator-autoseed --output salida.vm bpel
spec
```

Donde `salida.vm` sería la salida generada en formato de Apache Velocity. También se puede realizar de forma alternativa sin utilizar la opción, como sigue:

```
test-generator-autoseed bpel spec > salida.vm
```

Semilla aleatoria

La última opción disponible en esta herramienta es la opción `seed`. Mediante esta opción, podemos indicar una semilla de 16 Bytes (en codificación *UTF-8*, una cadena de 16 caracteres) la cual tomará el generador de números aleatorios para generar el conjunto de casos de prueba base. Se utilizaría de la siguiente forma:

```
test-generator-autoseed --seed semilla bpel spec
```

Siendo semilla una cadena de 16 caracteres cualquiera. Esta opción es útil cuando queremos comparar la efectividad de la técnica respecto de la generación aleatoria pura, por ejemplo, o para cuestiones de depuración comprobar que las modificaciones se realizan correctamente.

C. Manual del desarrollador

A continuación, se indican los pasos a seguir para incluir soporte para otros entornos y nuevos formatos de informe a TestGenerator-Autoseed.

C.1. Descarga del proyecto TestGenerator-Autoseed

Para comenzar con el desarrollo, es necesario descargar la última versión del proyecto de la forja *Redmine*. Para ello, se introduce el siguiente comando en la terminal[1]:

```
svn co https://neptuno.uca.es/svn/sources-fm/trunk/src/
test-generator-autoseed/
```

Una vez finalizado, se tendrá una copia de trabajo del repositorio en la que se podrá comenzar con el desarrollo.

C.2. Creación del proyecto para Eclipse

Una vez descargada la copia de trabajo, es necesario generar un proyecto para Eclipse, para facilitar la labor del desarrollo. Para ello, en el mismo directorio donde se descargó la copia, se ejecuta:

```
mvn eclipse:eclipse
```

Una vez realizado, se habrá generado un proyecto para Eclipse, el cual podrá ser importado en dicho IDE para comenzar a desarrollar.

[1] Es necesario tener instalado el sistema de control de versiones Subversion en la máquina.

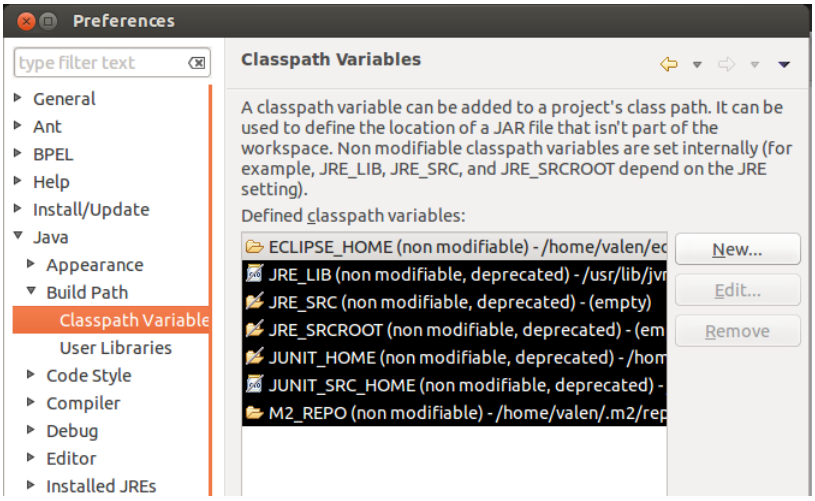


Figura C.1.: Variable M2_REPO en Eclipse

En Eclipse será necesario crear la variable de entorno *M2_REPO*, la cual es necesaria para resolver las dependencias del proyecto en el repositorio *maven*. Para ello, se acude a *Window* → *Preferences*, y allí a *Java* → *Build Path* → *Classpath Variables* (Ver figura C.1).

Se crea una nueva variable, con el nombre dado anteriormente, cuyo valor será el directorio donde se aloja el repositorio maven (normalmente `~/.m2/repository`).

Una vez está el entorno listo, se podrá comenzar a trabajar.

C.3. Incorporación de soporte para otros entornos

La metodología a seguir para dotar a TestGenerator-Autoseed de la capacidad de funcionar sobre otro entorno que no sea el de las composiciones WS-BPEL es la que veremos a continuación:

1. En el apartado de lectura de constantes, se implementa una nueva clase Java que implemente la interfaz *ConstReader*, dentro del paquete `reader.constants`, la cual permita leer constantes y variables relacionadas en el lenguaje destino.

2. En el apartado de lectura de variables, se implementa también otra clase Java, en este caso la cual debe realizar la interfaz *TypeReader*, dentro del paquete `reader.variables`.

3. Para realizar el chequeo de variables válidas, se crea una nueva clase Java que implemente la interfaz *TypeChecker*, en el paquete `check.algorithm`.

4. Para la generación de casos de prueba, nos basta con crear una clase Java que implemente a *AutoSeeder*, del paquete `generators`.

5. Por último, basta con extender las clases *AutomaticSeeding*, del paquete `algorithm`, y TestGeneratorAutoseedCommand, del paquete `cli`, para redefinir el control y los parámetros necesarios para el nuevo entorno.

Además de todo esto, se necesitará una batería de pruebas específicas para el nuevo entorno, que puede seguir el esquema de pruebas existente dentro de `src/test/java`.

C.4. Incorporación de nuevos formatos para el informe

La metodología a seguir para agregar nuevos formatos de salida para el informe es la siguiente:

1. Se implementa una clase Java que extienda a la clase *AbstractReportFormatter*, disponible en el paquete `report`, dentro de la carpeta `src/main/java`.

   El nombre de esta clase debe seguir el siguiente formato:

   [FORMATO]ReportFormatter

   Donde [FORMATO] es el nombre del formato que queremos agregar. La clase deberá de permanecer en el mismo paquete que *AbstractReportFormatter*.

2. Se implementan las pruebas unitarias empleando JUnit, dentro de las siguientes clases:

a) *CommandTest*. En esta clase se encuentran las pruebas de la CLI, y es necesario añadir una nueva prueba donde se solicite a la misma el formato de informe pedido. Esta clase está en el paquete `cli` dentro de la carpeta `src/test/java`.

b) *ReportTests*. En esta clase se encuentran las pruebas unitarias de generación del informe. Por lo tanto, se debe de incluir una nueva prueba donde se compruebe que el formato es el esperado. Esta clase está en el paquete `report` dentro de `src/test/java`.

A la hora de invocar el nuevo formato mediante la opción pertinente, se empleará el nombre indicado con anterioridad.

C.5. Ejecución de las pruebas unitarias

Una vez realizada la extensión que se desee, es necesario ejecutar las pruebas unitarias para verificar su correcto funcionamiento. Para ello existen dos opciones:

1. Desde Eclipse, se abre el fichero de casos de prueba, y se acude a Run $\rightarrow$ Run As $\rightarrow$ JUnit Test. El entorno procederá a ejecutar el conjunto de casos de prueba dado, indicando al finalizar si se han pasado con éxito. En el caso de haber algún caso de prueba no superado, se indica el motivo. (Ver figura C.2)

   De esta forma, se pueden ejecutar las pruebas individualmente, es decir, las pruebas del operador para una composición en concreto. Esto es más cómodo para el proceso de depuración del operador.

2. Desde el directorio `test-generator-autoseed`, se ejecuta:

   ```
   mvn test
   ```

   De esta forma, se ejecutan todos las pruebas unitarias, y se genera un informe detallado. Esto es más cómodo para verificar que todo funciona correctamente una vez concluye el desarrollo.

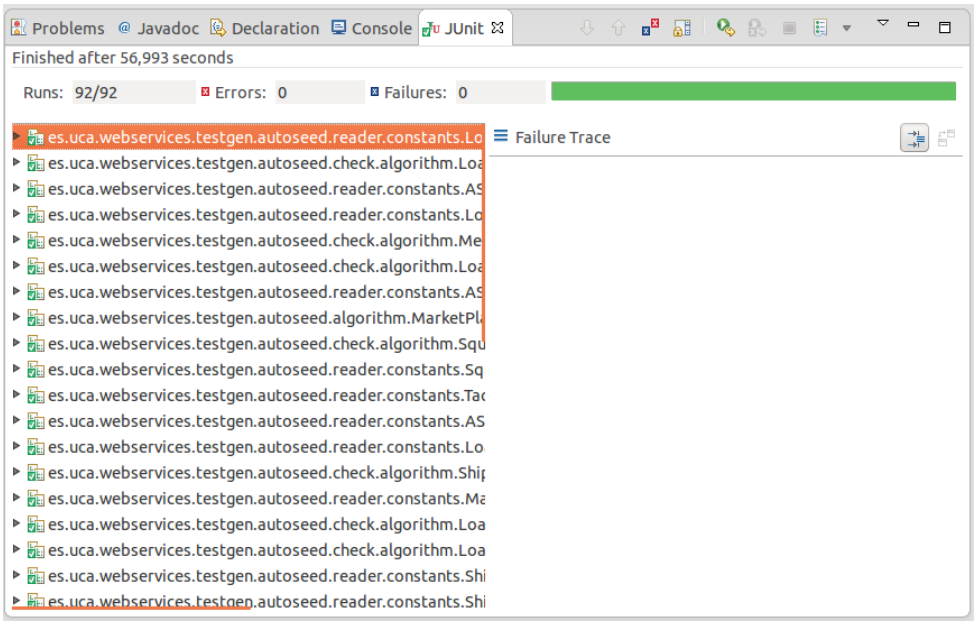


Figura C.2.: Ejecución de las pruebas unitarias en eclipse

C.6. Generación del ejecutable

Una vez concluido el proceso de prueba, se puede generar una nueva versión de **TestGenerator-Autoseed**, la cual contendrá la funcionalidad añadida. Para ello, será necesario generar un nuevo *.jar*, el cual se obtiene mediante la orden:

```
mvn package
```

De esta forma se obtiene un fichero en el directorio `test-generator-autoseed/target` cuyo nombre tiene la siguiente forma: *test-generator-autoseed-x.y.z-SNAPSHOT-dist.tar.gz*.

Descomprimiendo ese fichero en el directorio `~/bin` y sustituyendo la versión anterior, puede ejecutarse la nueva versión de **TestGenerator-Autoseed**:

```
test-generator-autoseed(argumentos)
```

Version 1.3, 3 November 2008

D. GNU Free Documentation License

D.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

D.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the

conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ‘Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

D.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions

whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 4.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

D.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

D.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 3 and 4 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.

- Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

D.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

D.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

D.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 4 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

D.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 5. Replacing Invariant Sections with translations requires special permission from their

copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 5) to Preserve its Title (section 2) will typically require changing the actual title.

D.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

D.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

D.12. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part

into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Bibliografía

[1] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, y Eugene Spafford. *Design of Mutant Operators for the C Programming Language*. Informe Técnico SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[2] N. Alshahwan y M. Harman. *Automated web application testing using search based software engineering*. En *ASE'11: 26th IEEE/ACM International Conference on Automated Software Engineering*, páginas 3–12. 2011.

[3] M. Alshraideh y L. Bottaci. *Search-based software test data generation for string data using program-specific search operators*. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[4] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, y Jérôme Siméon. *XML Path Language (XPath) 2.0*. Informe técnico, W3C, 2010. URL `http://www.w3.org/TR/xpath20/`.

[5] Paul V. Biron y Ashok Malhotra. *XML Schema Part 2: Datatypes*. Informe técnico, W3C, 2004. URL `http://www.w3.org/TR/xmlschema-2/`.

[6] Juan Boubeta-Puig. *Implementación de operadores para WS-BPEL 2.0*. Proyecto Fin de Carrera, Escuela Superior de Ingeniería, Universidad de Cádiz, 2010. URL `http://rodin.uca.es:8081/xmlui/handle/10498/9755`.

[7] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, y François Yergeau. *Extensible Markup Language (XML) 1.0* . Informe técnico, W3C, 2008. URL `http://www.w3.org/TR/xml/`.

[8] Sebastian Breier. *Extended Data-flow Analysis on BPEL Processes*. Proyecto Fin de Carrera, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, July 2008. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2726&engl=1`.

[9] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, y Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Informe técnico, W3C, 2007. URL `http://www.w3.org/TR/wsdl20`.

[10] Jenkins CI. *Jenkins Home Page*, Enero 2014. URL `https://jenkins-ci.org`.

[11] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. Informe técnico, Subversion, Junio 2004. URL `http://svnbook.red-bean.com/nightly/en/index.html`.

[12] Edsger W. Dijkstra. *Notes on structured programming*, capítulo Chapter I: Notes on structured programming, páginas 1–82. Academic Press Ltd., London, UK, UK, 1972. ISBN 0-12-200550-3.

[13] Antonio García Domínguez. *XMLEye Home Page*, Enero 2014. URL `https://neptuno.uca.es/redmine/projects/sources-fm/wiki/XMLEye`.

[14] Antonia Estero-Botaro, Juan Boubeta-Puig, Valentín Liñeiro-Barea, y Inmaculada Medina-Bulo. *Operadores de Mutación de Cobertura para WS-BPEL 2.0*. En *Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, tomo 4, páginas 355–368. 2012.

[15] Antonia Estero-Botaro, Francisco Palomo-Lozano, y Inmaculada Medina-Bulo. *Mutation Operators for WS-BPEL 2.0*. En *21th International Conference on Software & Systems Engineering and their Applications*. Paris, Francia, 2008.

[16] José-Luis Ezquerro-Casado. *Generador de conjuntos de pruebas paramétricos BPELUnit para composiciones WS-BPEL*. Proyecto Fin de Carrera, Escuela Superior de Ingeniería, Universidad de Cádiz, 2013. URL `http://rodin.uca.es/xmlui/handle/10498/15485`.

[17] The Apache Software Foundation. *Apache Maven Home Page*, Enero 2014. URL `https://maven.apache.org/`.

[18] The Apache Software Foundation. *Apache Velocity Template Language Reference*, 2014. URL `http://velocity.apache.org/engine/devel/vtl-reference-guide.html`.

[19] The Apache Software Foundation. *XMLBeans Home Page*, Enero 2014. URL `https://xmlbeans.apache.org/`.

[20] The Eclipse Foundation. *Eclipse Home Page*, Enero 2014. URL `https://www.eclipse.org/`.

[21] Martin Fowler. *Continous Integration*, 2006. URL `http://martinfowler.com/articles/continuousIntegration.html`.

[22] G. Fraser y A. Arcuri. *The Seed is Strong: Seeding Strategies in Search-Based Software Testing*. En *ICST'12: IEEE Fifth International Conference on Software Testing, Verification and Validation*, páginas 121–130. IEEE Computer Society, 2012.

[23] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Patrones de Diseño*. Addison-Wesley, 2003.

[24] Antonio García Domínguez, Antonia Estero Botaro, Juan José Domínguez Jiménez, Inmaculada Medina Bulo, y Francisco Palomo Lozano. *MuBPEL: una Herramienta de Mutación Firme para WS-BPEL 2.0*. En Antonio Ruiz y Luis Iribarne, editores, *Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos*, páginas 415–418. Almería, Spain, septiembre 2012. ISBN 978-84-15487-28-9. URL `http://sistedes2012.ual.es/sistedes/jisbd`.

[25] Y. Jia y M. Harman. *An Analysis and Survey of the Development of Mutation Testing*. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011. ISSN 0098-5589. doi:{10.1109/TSE.2010.62}.

[26] Cristina Jiménez-Gavilán. *Analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas*. Proyecto Fin de Carrera, Universidad de Cádiz, 2011. URL `http://hdl.handle.net/10498/11695`.

[27] JUnit. *Resources for Test Driven Development*, Enero 2014. URL `http://www.junit.org/`.

[28] K. N. King y A. Jefferson Offutt. *A FORTRAN Language System for Mutation-based Software Testing*. *Software – Practice and Experience*, 21(7):685–718, 1991.

[29] Valentín Liñeiro-Barea. *Proyecto Fin de Carrera: Operadores de Mutación de Cobertura para WS-BPEL 2.0*. Proyecto Fin de Carrera, Escuela Superior de Ingeniería, Universidad de Cádiz, Septiembre 2011. URL `http://rodin.uca.es/xmlui/handle/10498/15759`.

[30] Valentín Liñeiro-Barea. *Respositorio de código fuente de la herramienta TestGenerator-Autoseed*, Enero 2014. URL `https://neptuno.uca.es/svn/sources-fm/trunk/src/test-generator-autoseed/`.

[31] Daniel Luebke y Antonio García-Domínguez. *BPELUnit - The Open Source Unit Testing Framework for BPEL*, Enero 2014. URL `http://bpelunit.github.io/`.

[32] Yu-Seung Ma, Jeff Offutt, y Yong-Rae Kwon. *MuJava: An Automated Class Mutation System*. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[33] Brent Matzelle y Alexander Mueller. *Página del proyecto RapidSVN*, Enero 2014. URL `http://rapidsvn.tigris.org/`.

[34] Glen McCluskey. *Using Java Reflection*. Informe técnico, Oracle, 1998. URL `http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html`.

[35] Jan Mendling, Kristian Bisgaard Lassen, y Uwe Zdun. *On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages*. *IJBPIM*, 3:96–108, 2008.

[36] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, Segunda edición, 2004.

[37] OASIS. *Web Services Business Process Execution Language 2.0*, 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[38] A. Jefferson Offutt, Jeff Voas, y Jeff Payn. *Mutation Operators for Ada*. techreport ISSE-TR-96-09, George Mason University, Fairfax, Virginia, 1996.

[39] Oracle. *Java 6 overview*, Septiembre 2011. URL `http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html`.

[40] Miguel Ángel Pérez-Montero. *Generador de casos de prueba basado en estrategias personalizables*. Proyecto Fin de Carrera, Universidad de Cádiz, 2013. URL `http://hdl.handle.net/10498/15486`.

[41] Emmanuel Rodriguez. *Xacobeo Home Page*, Enero 2014. URL `https://code.google.com/p/xacobeo/`.

[42] Ian Sommerville. *Ingeniería del Software: Un Enfoque Práctico*. Pearson, séptima edición, 2005.

[43] Sonatype. *Sonatype Nexus Repository Manager*, Enero 2014. URL `http://www.sonatype.org/nexus/`.

[44] JOptSimple Development Team. *JOptSimple Home Page*, Enero 2014. URL `http://pholser.github.io/jopt-simple/`.

[45] Javier Tuya, Isabel Ramos Román, y Javier Dolado Cosín. *Técnicas cuantitativas para la gestión en la ingeniería del software*, capítulo 14, páginas 310–311. NetBiblo, 2007.

[46] Javier Tuya, María J. Suárez-Cabal, y Claudio de la Riva. *Mutating database queries*. *Information and Software Technology*, 49(4):398–417, 2007.

[47] Grupo UCASE. *TestSpec format*, Enero 2014. URL `https://neptuno.uca.es/redmine/projects/sources-fm/wiki/Spec_format`.

[48] Grupo UCASE. *Wiki del proyecto MuBPEL*, Enero 2014. URL `https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL`.

[49] Grupo UCASE. *WS-BPEL Composition Repository*, Enero 2014. URL `https://neptuno.uca.es/redmine/projects/wsbpel-comp-repo/wiki`.

[50] Grupo UCASE. *WS-BPEL Testing Tools Project*, Enero 2014. URL `https://neptuno.uca.es/redmine/projects/sources-fm`.

[51] Kai Willadsen. *Meld Home Page*, Enero 2014. URL `http://meldmerge.org/`.

[52] Xuehong Yang, Junfei Huang, y Yunzhan Gong. *Static data flow analysis and anomalies detection for BPEL*. En *Test and Measurement, 2009. ICTM '09. International Conference on*, tomo 2, páginas 18–21. 2009. doi:{10.1109/ICTM.2009.5413080}.